EFFICIENT MATRIX-AWARE RELATIONAL QUERY PROCESSING IN BIG DATA SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Yongyang Yu

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Walid G. Aref, Chair

Department of Computer Science

Dr. Elisa Bertino

Department of Computer Science

Dr. Sunil Prabhakar

Department of Computer Science

Dr. Alex Pothen

Department of Computer Science

Approved by:

Dr. Voicu S. Popescu

Head of the School Graduate Program

This dissertation is dedicated to my parents for their love, endless support and encouragement.

ACKNOWLEDGMENTS

I am really thankful to my academic advisor, Dr. Walid G. Aref, who guides me to explore and appreciate the beauty of database systems. Dr. Aref is a respectable and brilliant researcher, who always pushes me to think deeper, and shares his own research experience. He is open-minded and willing to listen to new ideas, always offering positive feedbacks whenever I propose novel research topics, without which I will never have the chance to accomplish my dissertation. He is such a diligent professor that I will never forget the days when we have revised my papers together late until the middle of the nights, even if he was in poor health condition. It is great fortune for me to work with this excellent person and benefit from his perspective.

This dissertation is the result of collaboration with a number of great collaborations. Chapter 2 is the joint work with Mingjie Tang, Walid G. Aref, Qutaibah Malluhi, Mostafa Abbas, and Mourad Ouzzani [1]. Chapter 3 extends the system developed in Chapter 2 for efficient processing queries that involve both matrix and relational operations. Chapter 4 demonstrates the performance of the proposed system with complex machine learning queries in deep learning applications. Both chapters are joint works with with Mingjie Tang and Walid G. Aref [2, 3].

I am very pleased to invite Dr. Elisa Bertino, Dr. Sunil Prabhakar, and Dr. Alex Pothen to serve on my final exam committee. Their suggestions improve my dissertation significantly. Beyond direct collaborators on my dissertation, many friends have contributed to my graduate work, and made Purdue an unforgettable experience. My labmates at Purdue database research group include Mingjie Tang, Ahmed R. Mahmood, and Amgad M. Madkour always offer their help and feedbacks to enhance this work. Shandian Zhe, Weihang Wang, Dong Su, Shuxian Jiang, Xuejiao Kang, Xin Cheng, Zhengyi Zhang, Mu Wang, Jiasen Yang, Yifan Yang, Xi Tan, and Xiao

Zhang were great fun to hang out with and fantastic people for great inspirations on some neat ideas.

Last but not the least, I want to show my gratitude to my father Tianqing Yu, and mother Lijuan Wang for their unwavering support throughout my Ph.D. study. Without their support, I cannot accomplish this work.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Yu, Yongyang Ph.D., Purdue University, December 2018. Efficient Matrix-aware Relational Query Processing in Big Data Systems. Major Professor: Walid G. Aref.

In the big data era, the use of large-scale machine learning methods is becoming ubiquitous in data exploration tasks ranging from business intelligence and bioinformatics to self-driving cars. In these domains, a number of queries are composed of various kinds of operators, such as relational operators for preprocessing input data, and machine learning models for complex analysis. Usually, these learning methods heavily rely on matrix computations. As a result, it is imperative to develop novel query processing approaches and systems that are aware of big matrix data and corresponding operators, scale to clusters of hundreds of machines, and leverage distributed memory for high-performance computation. This dissertation introduces and studies several matrix-aware relational query processing strategies, analyzes and optimizes their performance.

The first contribution of this dissertation is MatFast, a matrix computation system for efficiently processing and optimizing matrix-only queries in a distributed in-memory environment. We introduce a set of heuristic rules to rewrite special features of a matrix query for less memory footprint, and cost models to estimate the sparsity of sparse matrix multiplications, and to distribute the matrix data partitions among various compute workers for a communication-efficient execution. We implement and test the query processing strategies in an open-source distributed dataflow engine (Apache Spark).

In the second contribution of this dissertation, we extend MatFast to MatRel, where we study how to efficiently process queries that involve both matrix and relational operators. We identify a series of equivalent transformation rules to rewrite

a logical plan when both relational and matrix operations are present. We introduce selection, projection, aggregation, and join operators over matrix data, and propose optimizations to reduce computation overhead. We also design a cost model to distribute matrix data among various compute workers for communication-efficient evaluation of relational join operations.

In the third and last contribution of this dissertation, we demonstrate how to leverage MatRel for optimizing complex matrix-aware relational query evaluation pipelines. Especially, we showcase how to efficiently learn model parameters for deep neural networks of various applications with MatRel, e.g., Word2Vec.

1 INTRODUCTION

In the big data era, data exploration aims at establishing correlations between things we do not know that may lead to new possibilities, unlike business intelligence (BI) systems where one knows what information she is looking for and designs systems to deliver the specific types of information, e.g., sum aggregation on a certain attribution in a report. Big data also exhibits quite different characteristics than traditional business data, known as 4V properties, i.e., Volume, Velocity, Variety, and Veracity [4, 5].

Volume represents the amount of data, such as Terabyte (TB, 2^{40} bytes) or Petabyte (PB, 2^{50} bytes). With the advent of Internet of Things (IoT) [6], data are generated much faster than ever before. For instance, about 2.5 quintillion bytes of data are created each day, and this speed is predicted to increase exponentially during the next decade according to International Data Corporation (IDC). At Facebook, daily active users upload 350 million photos, and more than 500 TB data are generated per day. These facts indicate a big data system must be able to handle tremendous volumes of data as inputs of typical workloads.

Velocity reflects the speed of data generation, update, or processing. Modern big data systems accept sources with high data generation rate. For example, generating 100 TB text data in 5 hours implies the generation rate of 20 TB/hour. In addition, a lot of big data applications require real-time data updates. For example, in a social network site, e.g., Facebook and Twitter, the social graph data is continuously updating when a friend or follower relationship is established or dropped. Furthermore, in streaming systems, data streams continuously arrive and the big data system must be able to process the data in real-time to keep up with the arrival speed. Therefore, it is very challenging for a big data system to reflect data generation rates, update frequencies, and processing speed.

Variety indicates the range of data types and sources. The fast development of IoT and remote sensing gives birth to a diversity of data types, such as structured data (e.g., relational tables), unstructured data (e.g., text, graph, images, audios, and videos), and semi-structured data (e.g., web logs, customer reviews, and resumes). Hence, it is imperative for a big data system to support a spectrum of data types including structured, semi-structured, and unstructured data, as well as representative data sources, such as table, text, stream, and graph.

Veracity [7] denotes if the data used in the query processing conforms to the inherent and important characteristics of the raw data. Sometimes, it is too expensive to process the raw data in huge volumes as there may not exist enough compute resources. Data sketch [8] is a promising technique to capture the main characteristics of the raw input with less data. Data veracity is important to guarantee the credibility of the results from a big data system.

Besides the 4V properties of big data, the workflows of queries also exhibit different features in a big data system from those of traditional BI queries. Data scientists and analysts often need to analyze large volumes of data in diverse applications, such as self-driving cars, social network analysis, web-search, online advertisement bidding, and recommender systems. Most of queries in these driving applications are expressed using machine learning (ML) models, e.g., principle component analysis (PCA) [9], collaborative filtering (CF) [10], and linear regression (LR) [11], that involve linear algebra operations and heavy matrix computations as building blocks. Additionally, many network analysis algorithms are expressed by matrix operations as well, e.g., PageRank, betweenness centrality, and spectral clustering [12]. Recently, tensor factorization [13] has become a popular model to capture relationships among multiple entities, which also extensively relies on matrix computations. Thus, it is of great importance for these models to have access to an efficient, scalable, and matrix-aware relational data system.

1.1 Motivation

The large amount of data accumulated from IoT, remote sensing, online social networks, and computational sciences calls for advanced matrix-aware relational query processing techniques. Consider the following application scenarios for matrix-aware relational query processing.

Motivating Scenario 1: Recommender System

Making recommendations to potential customers is a million-dollar business in marketing, e.g., Netflix prize[1] for movie recommendations. Gaussian Non-negative Matrix Factorization (GNMF) [14], is a widely used ML model for clustering documents and modeling topics of massive text data. The input to GNMF is a $d \times w$ document-term Matrix $\mathbf{V}$, where d corresponds to the number of documents, and w corresponds to the number of terms. Each cell V_{ij} records the frequency of term j in document i. To predicate an unseen term in a certain document, GNMF assumes that Matrix $\mathbf{V}$ can be characterized by p hidden topics such that $\mathbf{V}$ can be factorized into the multiplication of two hidden factor Matrices $\mathbf{W}_{d\times p}$ and $\mathbf{H}_{p\times w}$, i.e., $\mathbf{V} \approx \mathbf{W} \times \mathbf{H}$. In real-world applications, the number of topics p is chosen between 50 and 200. Typically, d and w are much larger than p. For example, GNMF could be applied to the Netflix contest dataset [15], where $d = 480,189$ and $w = 17,770$.

The estimations for Matrix $\mathbf{W}$ and $\mathbf{H}$ could be conducted in an iterative manner.

```
1 val p = 100 // number of topics
2 val V = loadMatrix("in/V") // read matrix
3 var W = RandomMatrix(V.nrows, p)
4 var H = RandomMatrix(p, V.ncols)
5 val max_iteration = 50
6 for (i <- 0 until max_iteration) {
7     H = H * (W.t %*% V) / (W.t %*% W %*% H)
8     W = W * (V %*% H.t) / (W %*% H %*% H.t)
```

[1] https://web.archive.org/web/20090924184639/http://www.netflixprize.com/community/viewtopic.php?id=1537

```
}
```

Code 1.1: GNMF algorithm in Scala

Code 1.1 illustrates the Scala source code for evaluating Matrix **W** and **H**, where "$*$" denotes matrix element-wise multiplication, "$\%*\%$" represents matrix-matrix multiplication, and "$\mathbf{A}.t$" indicates the transpose of matrix **A**. In contrast to queries in a relational database, this query exposes several specific features: *big matrix data retrieval*, *common matrix operators*, and *iterative execution*.

Motivating Scenario 2: Web Search

The World Wide Web (WWW) hosts trillions of hyperlinked documents. One of the most important tasks a search engine, e.g., Google, faces every day is how to rank the relevant webpages according to a user's query. PageRank [16, 17] is a link analysis model and it assigns a numerical weight to each document, with the purpose of measuring its relative importance within WWW. PageRank models the whole WWW as a graph, where each document serves a a vertex and the hyperlink between a pair of documents serves a directed edge in the graph.

```
val A = loadMatrix("in/A") // adjacency matrix
val d = rowSum(A) // out-going degrees of vertices
val P = A.t %*% diag(1.0 / d) // transition matrix
var v = ones(A.nrows, 1) // jumping vector
v = v / sum(v) // normalization
val alpha = 0.85 // damping factor
var x = v
val max_iteration = 100
for (i <- 0 until max_iteration) {
    x = alpha * P %*% x + (1.0 - alpha) * v
}
```

Code 1.2: PageRank algorithm in Scala

Code 1.2 illustrates the computation steps for PageRank vector $\mathbf{x}$ in Scala code. The input to PageRank is the adjacency matrix $\mathbf{A}$ of WWW, which is a square matrix of 30 trillion rows and columns. In order to compute the transition matrix $\mathbf{P}$, a sum aggregation along the row dimension is conducted for the out-going degrees of all the vertices. Next, Matrix $\mathbf{P}$ is computed by a matrix-matrix multiplication between the transpose of the adjacency matrix $\mathbf{A}$ and the diagonal matrix of the inverse of the out-going degree vector. The PageRank vector $\mathbf{x}$ is evaluated in an iterative manner. Besides common matrix operators and iterative execution, this query also exposes *relational operations* on big matrix data, e.g., sum aggregation along the row dimension. It is worth noting that the size of the entire web graph is so massive that single-node solutions cannot process the data efficiently. It is necessary to have a *distributed* engine for executing such queries.

Motivating Scenario 3: Bioinformatics – eQTL

With the development of biology and bioinformatics technologies, genetic diagnosis has emerged as a promising mechanism for modern clinical medicine. It allows the analysis of chromosomes, proteins, and certain metabolites in order to detect heritable disease-related genotypes, mutations, phenotypes, or karyotypes for clinical purposes [18]. Expression quantitative trait loci (eQTLs) are genomic loci that explain all or a fraction of variation in expression levels of mRNAs [19]. eQTL analysis is a widely used method to find out how a given genotype at a particular QTL affects gene expression at the certain locus. Several statistical methods have been explored for eQTL analysis. kruX [20] is an efficient matrix-based tool for calculating the non-parametric test statistics, Kruskal-Wallis, to identify eQTL.

---

1 `Load dense genotype matrix` $\mathbf{G}$

2 `Generate index matrix` $I_i(m,k) = 1$ `if` $G(m,k) = i$

3 `Compute vector` $\mathbf{N}_i$`, where` $N_i(m) = \sum_{k=1}^{K} I_i(m,k)$

4 Compute $S_i(n,m) = \sum_{k=1}^{K} R(n,k) * I_i(m,k) = (\mathbf{R} \times \mathbf{I}_i^T)(n,m)$, where $R(n,k)$ records the rank for the nth trait in the ith genotype group of the mth marker

5 Compute $S(n,m) = \frac{12}{K(K+1)} \sum_{i=0}^{\ell} \frac{S_i(n,m)^2}{N_i(m)} - 3 * (K+1)$

Code 1.3: Computation steps of kruX

Code 1.3 demonstrates the computation steps of kruX. The input to kruX is a dense genotype Matrix $\mathbf{G}$. To compute the index matrices, e.g., $\mathbf{I}_0$, a compute engine needs efficient *search on matrix data* to locate the positions of certain values. Furthermore, the computation of Matrix $\mathbf{S}$ requires *sum aggregation* on the result of matrix element-wise divisions. It is also worth noting that the computation steps are CPU intensive, since matrix computations consume much more CPU cycles than traditional BI queries. Therefore, it is necessary to have an *in-memory* compute engine for executing such computation intensive queries.

Motivating Scenario 4: Natural Language Processing – Word2Vec

In natural language processing, semantic similarity is a metric defined over a set of documents or terms to measure the distance between them based on the likeness of their meaning or semantic contend as opposed to similarity which are estimated by syntactical representations. For example, "Madrid" has a smaller semantic distance to "Spain" than "Portugal". Recently, Word2Vec [21] is proposed as a group of related models to produce word embeddings [22], which are leveraged to compute the semantic similarity between words in a corpus. These models are two-layer neural networks [23] that are trained to reconstruct linguistic contexts of words. The input to Word2Vec is a corpus of text, and it produces a vector space, usually of several hundred dimensions, where each unique word in the corpus is assigned with a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located closer in the space.

Let us take a look at skip-gram model for the detailed computation steps. Given a trained skip-gram model, the input vectors are one-hot encoded[2]. To predict the adjacent words around an input $\mathbf{x}$, the hidden layer vector is evaluated as $\mathbf{h} = \mathbf{x}^T \times \mathbf{W}$, where $\mathbf{W}$ is the weight matrix between the input layer and the hidden layer. Similarly, the inputs to each of the $C \times V$ output nodes is computed by the weighted sum of its inputs, e.g., the input to the j-th node of the c-th output word is $u_{c,j} = v'^T_{w_j} \times \mathbf{h}$. Finally, the probability that the output of the j-th node of the cth output word is $y_{c,j} = exp(u_{c,j}) / \sum_{j'=1}^{V} exp(u_{j'})$.

```
1 Training set S = {y_1, ..., y_c}
2 Initialize weight matrix W_{V×N} and hidden layer weight matrix U_{N×V}
3 for x ∈ S {
4     U_ij = U_ij − η · Σ_{c=1}^{C} (y_{c,j} − t_{c,j}) · h_i
5     W_ij = W_ij − η · Σ_{j=1}^{V} Σ_{c=1}^{C} (y_{c,j} − t_{c,j}) · U_ij · x_j
6 }
```

Code 1.4: Back propagation of skip-gram model with stochastic gradient descent

Code 1.4 illustrates the back propagation steps for model training of skip-gram. $t_{c,j}$ is an indicator variable, where $t_{c,j} = 1$ if the jth node of the cth true output word is equal to 1. η is the step length of the stochastic gradient descent. To accelerate the back propagation procedure, the negative sampling technique [24] is adopted to update only a sample of weights in matrix $\mathbf{W}$ and $\mathbf{U}$. Both prediction and back propagation of Word2Vec models rely heavily on common *matrix operations* and *iterative executions*. Furthermore, optimization strategies, e.g., negative sampling, require *common relational operations* on matrix data, such as relational selection on the negative words. Thus, it is very important that a compute engine supports both common matrix and relational operations on big matrix data.

Motivating Scenario 5: GIS – Raster Data Analysis

[2] https://en.wikipedia.org/wiki/One-hot

A geographic information system (GIS) is a system designed to integrate, store, manipulate, analyze, manage, and display spatial or geographical data [25]. Modern GIS technologies utilize digitalized data and helps decision-making in various applications, such as urban planning, transport/logistics, insurance, telecommunications, and business [26]. With the development of remote sensing techniques, more and more ortho-rectified images are collected every day from space satellites, aircraft, camera drones [27], intelligent robots [28]. These images are typically stored using two broad methods: raster images and vectors. The images facilitate numerous advanced analytics in map-based applications and scientific research. For example, map overlay is a useful GIS operation that superimposes multiple datasets for the purpose of identifying relationships between them [29]. Especially, it could be leveraged to answer queries, such as "What landuse is on top of what soil type?" and "What roads are within what counties?".

```
1 val A = loadRaster("in/A")
2 val B = loadRaster("in/B")
3 val C = join(A, B, "A.rid = B.rid and A.cid = B.cid")
4 select(C, 'temp > 20 and type = "AG"')
```

Code 1.5: Map overlay operation in Scala

Code 1.5 demonstrates computing the overlay between two maps in raster format. The raster format could be represented by a matrix naturally. In addition, the overlay operation relies heavily on *relational joins*, as it combines characteristics from numerous layers into a single layer. Common *relational selections* are utilized to pick specific cells and attributes from the overlayed matrix.

1.2 Challenges in Matrix-aware Relational Query Processing

There is a wide range of applications that rely on matrix-aware relational query processing. These include databases, data mining, machine learning, bioinformatics,

and GIS. From aforementioned driving application scenarios, we summarize important challenges that need to be addressed for a big data system to support efficient matrix-aware relational query processing.

- **Big Matrix-data Storage, Retrieval, and Search**

  The 4V properties of big data imply large volumes of matrix data from various data sources, e.g., scientific report, satellite image, text, software logs. Often, only a subset of the entire database or data warehouse is utilized for answering a certain query. For commercial and scientific applications, the raw input data contains errors and missing values for some attributes. This requires efficient matrix-data storage, retrieval, and search to filter out noisy data and provide data prepocessing functionality for subsequent advanced analytical operations.

- **Supporting Common Matrix Operators**

  A plethora of applications require a number of different kinds of matrix operators, e.g., matrix transpose, matrix-scalar operators, matrix element-wise operators, matrix-matrix multiplications. These operators are organized in a pipeline, where the output of an operator is fed as input to another. It is necessary for a big data system to support common matrix operators and make extensions for new application specifications.

- **Supporting Queries with Both Relational and Matrix Operations**

  Complex analytics require multiple stages to accomplish a single task, and each stage may be composed of different types of operations, e.g., selecting a subset from a data source using relational selection and performing matrix operations on the subset; partitioning the input matrix into multiple sections for cross-validation [23] when training an ML model; obtaining average users' rating on a movie after predicting the missing values in the user-movie rating matrix. Thus, it is necessary for a big data system to support both relational and matrix operators efficiently for various applications scenarios.

- **Supporting Iterative Execution**

  Traditional SQL query execution is acyclic in nature. Matrix-aware relational query execution require iterative executions, since many ML models and matrix algorithms need multiple iterations before converge to the optimal solution. The data flows between different operators exhibit strong data dependency during iterative executions. A temporary computed solution often needs to be cached for repetitively access. Therefore, it is very important for a big data system to support efficient iterative executions.

- **Optimizing Queries That Involve Both Relational and Matrix Operators**

  When a query involves both relational and matrix operators, the optimization for query processing is non-trivial. Existing optimization techniques in RDBMSs have been extensively studied for relational data and operators. However, these techniques need to be studied thoroughly before they could be applied to matrix data and corresponding operators. For example, relational selection predict pushdown is a common optimization technique to reduce input data size before a join operation. A direct application of the pushdown mechanism to a query that involve both relational and matrix operators may produce erroneous result when the input matrices are sparse. Thus, carefully designed and studied optimization techniques are called for queries that involve both relational and matrix operators.

- **Supporting In-memory Query Processing in Distributed Big Data Systems**

  Recently, there has been a trend for in-memory database systems, which reduces I/O costs significantly by storing all the data in memory. Furthermore, matrix computations are inherently CPU and memory intensive. Disk-based distributed big data systems are *not* good candidates for matrix queries, e.g., Hadoop [30], as intermediate computed results have to be written to and read

from disks repeatedly for iterative executions. Therefore, it is essential to create an in-memory system that efficiently support matrix-aware relational query processing by leveraging the distributed memory inside a cluster of computers.

- **Communication Overhead in the Distributed Setup**
  A salient issue for distributed matrix-aware relational query processing is the extensive communication overhead when transferring data from different compute nodes over the network. Load-balanced data partitioning schemes on big matrix data may not be efficient, e.g., round-robin data partitioning for matrix-matrix multiplications. Due to iterative executions, data dependency also needs to be considered when storing intermediate matrices in distributed memory. Thus, it is crucial for a big data system to have little communication overhead for matrix-aware relational query processing.

1.3 Hypothesis of the Dissertation

Based on the observations and challenges from motivating examples, we claim: *"it is possible to build a computation- and communication-efficient matrix-aware relational query processor and optimizer for queries that combine both relational and matrix-based operators."*

1.4 Summary of Contributions

1. Query Processing for Queries with Matrix-only Operators in Big Data Systems

   - We develop MatFast, a matrix computation system for efficiently processing and optimizing queries with matrix-only operators in a distributed in-memory environment.
   - We introduce a cost model to accurately estimate the sparsity of sparse matrix multiplications, and propose heuristic rules to rewrite special features of a query with matrix-only operators for mitigating memory footprint.

- We investigate several popular matrix data partitioning schemes, and derive the conversion overhead between different partitioning schemes. In addition, we introduce a second cost model to distribute the matrix data partitions among a number of compute workers for a communication-efficient execution plan.
- We conduct an extensive experimental study of MatFast against state-of-the-art distributed matrix computation systems using real and synthetic datasets with various ML applications. The experiments illustrate the effectiveness of our proposed cost models for reducing memory consumption and communication overhead. Experimental results illustrate up to an order of magnitude enhancement in performance.

2. Query Processing for Queries That Involve Both Relational and Matrix Operators in Big Data Systems

   - We develop MatRel, a system for efficient query processing for queries that involve both relational and matrix operators in a distributed in-memory environment.
   - We identify a series of equivalent transformation rules to rewrite a logical plan when both relational and matrix operations are present. Especially, we extend the relational predicate pushdown heuristic to a mixture of relational selection/aggregation and matrix operations for reducing computation overhead.
   - We formally define the join operator on matrix data based on different variants of the join predicates, and propose a number of optimization techniques to enhance runtime cost. By identifying sparsity-preserving merge functions and adopting the Bloom-join strategy when the join predicate contains matrix entries, MatRel is able to generate efficient execution plans on sparse matrices. We introduce a cost model to distribute matrix

data among various compute workers for communication-efficient evaluation of relational join on big matrix data.

- We extend MatFast to MatRel, and realize the introduced query processing and optimization techniques. We use the developed prototype system to conduct a number of evaluations on both real and synthetic datasets. We compare MatRel against state-of-the-art distributed matrix computation systems and an array database. Experimental results illustrate up to two orders of magnitude enhancement in performance.

3. Optimizing Complex Matrix-aware Relational Query Evaluation Pipelines (with Deep-learning as a Driving Application)

    - We demonstrate the domain specific language and expressive programming interface of MatRel in the DataFrame API for evaluating complex ML pipelines.
    - We illustrate the superiority of MatRel in evaluating deep-learning models, e.g., Word2Vec. We discuss the detailed computation steps of the predication procedure and the back propagation for model training. By examining the core operation required by deep-learning models, we show how MatRel could be leveraged to reduce the computation and communication overhead. Experimental results on real datasets show enhancements in performance by up to an order of magnitude over existing deep-learning platforms.

1.5 Dissertation Outline

We have published parts of the work presented in this dissertation [1–3]. The work on query processing and optimizations for queries with matrix-only operators is presented in [1]. The study of query processing for queries that involve both relational

and matrix operators is presented in [2]. The demonstration of optimizing complex matrix-aware relational query evaluation pipeline is presented in [3].

The remaining of the dissertation is organized as follows. Chapter 2 presents the preliminaries and common notations for matrices and tensors. Chapter 3 introduces query processing and optimizations for queries with matrix-only operators. Chapter 4 covers query processing and optimizations for queries that involve both relational and matrix operators. Chapter 5 discusses optimizing complex matrix-aware relational query evaluation pipelines, especially we demonstrate how to leverage the techniques and the system we've developed in previous chapters for deep-learning applications. Chapter 6 concludes this dissertation and discusses possible areas for future work.

2 PRELIMINARIES AND NOTATIONS

In this chapter, we briefly introduce the common notations used in this dissertation.

2.1 Notations

We follow the convention that a bold, upper-case Roman letter, e.g., $\mathbf{A}$, denotes a matrix and regular Roman letter with subscripts, e.g., A_{ij} represent single elements in a matrix. A column vector is written in bold, lower-case Roman letter, e.g., $\mathbf{x}$. A scalar is written in lower-case Greek letter, e.g., β. A block matrix is written as $\mathbf{A}_{ij}$, where i and j are the row-block index and column-block index. (X_{ij}) represents a matrix with element X_{ij} at the i-th row and the j-th column. A bold, upper-case calligraphic Roman letter, e.g., $\boldsymbol{\mathcal{A}}$, denotes a tensor, and regular Roman letter with subscripts, e.g., $\mathcal{A}_{ijk}$ or $\mathcal{A}_{ijk\ell}$, represents an element in a 3rd- or 4th-order tensor [31]. The *order* of a tensor is the number of dimensions, and it can be inferred from the context when we discuss corresponding operators in Chapter 4.

2.2 Matrix operators

To enable matrix-aware relational query processing, a big data system should be able to support both common unary and binary matrix operators. For unary matrix operator, we have matrix transpose $\mathbf{B} = \mathbf{A}^T$, $B_{ij} = A_{ji}, \forall i, j$. Binary operators includes matrix-scalar addition, $\mathbf{B} = \mathbf{A} + \beta$, $B_{ij} = A_{ij} + \beta$; matrix-scalar multiplication, $\mathbf{B} = \mathbf{A} * \beta$, $B_{ij} = \beta * A_{ij}$; matrix-matrix addition, matrix-matrix element-wise multiplication, matrix-matrix element-wise division, $\mathbf{C} = \mathbf{A} \star \mathbf{B}$, $C_{ij} = A_{ij} \star B_{ij}$, where $\star \in \{+, *, /\}$; and matrix-matrix multiplication, $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, $C_{ij} = \sum_k A_{ik} * B_{kj}$. In

our code snippet, we use $\mathbf{A}.t$ to denote the transpose for matrix $\mathbf{A}$. For matrix-matrix multiplications, $\mathbf{A}\%*\%\mathbf{B}$ has the same meaning as $\mathbf{A} \times \mathbf{B}$.

2.3 Relational operators

To enable relational query processing on big matrix data, a big data system should support basic relational operators, such as selection, projection, aggregation, and join. As matrix data is two dimensional, the relational predicates for these common operators may involve both row and column dimensions, and entry values. We adopt common notations from relational algebra [32]. The relational selection is represented as $\sigma_{\theta(RID,CID,val)}(\mathbf{A})$, where θ is a propositional formula on the dimensions and entries of Matrix $\mathbf{A}$. A relational projection on a matrix is denoted as $\Pi_{\tau(RID,CID,val)(\mathbf{A})}$, where τ is a subset of all the attributes $\{RID, CID, val\}$. An aggregation is formally defined as $\Gamma_{\rho,dim}(\mathbf{A})$, where ρ is the name of the aggregate function, e.g., `sum`, and dim indicates the chosen dimension, e.g., row or column. A relational join is a binary operator, defined as $\mathbf{A} \bowtie_{\gamma,f} \mathbf{B}$, where γ is the join predicate, and f is the user-defined merge function that takes two matching entries and outputs a merging value. A thorough discussion of the parameters about each operator is presented in Chapter 4.

3 QUERY PROCESSING FOR QUERIES WITH MATRIX-ONLY OPERATORS IN BIG DATA SYSTEMS

3.1 Introduction

In the era of big data, data scientists and analysts often need to analyze large volumes of data in a diverse array of application such as BI, self-driving cars, social network analysis, web-search, online advertisement bidding, and recommender systems. Most of the algorithms in these applications are expressed using machine learning (ML) models, e.g., principle component analysis (PCA), collaborative filtering (CF) and linear regression (LR), that involve linear algebra operations and heavy matrix computations as building blocks. Furthermore, many network analysis algorithms are expressed using matrix operations, e.g., PageRank, betweenness centrality, and spectral clustering [12]. Recently, tensor factorization [13] has become a popular model to capture relationships among multiple entities, which also extensively relies on matrix computations. Thus, it is important for these models to have access to an efficient and scalable execution engine for matrix computations. The advent of MapReduce [33] has spurred numerous distributed matrix computation systems, e.g., HAMA [34], Mahout [35], and SystemML [36]. These systems not only provide comparable compute efficiency to widely used scientific platforms [36], e.g., R [37], but also offer better scalability and fault-tolerance. However, these systems suffer from two main shortcomings. First, they are unable to reuse intermediate data [38]. The inability to efficiently leverage intermediate data greatly impedes the performance of further data analysis with matrix computations. In addition, these systems do not leverage the power of distributed memory offered by modern hardware.

One promising way to address the above challenges is to develop an efficient execution engine for large-scale matrix computations based on an in-memory distributed

cluster of computers. Apache Spark [38] is a computation framework that allows users to work on distributed in-memory data without worrying about the data distribution or fault-tolerance. Recently, a variety of Spark-based systems for matrix computations have been proposed, e.g., MLI [39], MLlib [40], and DMac [41]. Although addressing several challenges in distributed matrix computation processing, none of the existing systems leverage some of the special features of matrix programs to generate efficient partitioning schemes for matrix data at both the input and intermediate stages. The special features we are referring to, and which are prevalent in ML algorithms, include sparse matrix chain multiplications, low-rank matrix updates, and invariant expressions in loop structures. Since matrix computations are inherently memory intensive, an execution engine that cannot leverage these special features will overwhelm the hardware capacity.

For illustration, consider Gaussian Non-negative Matrix Factorization (GNMF) [14], a widely used ML model for clustering documents and modeling topics of massive text data. Code 1.1 shows the compute steps of GNMF. GNMF assumes that Matrix $\mathbf{V}$ can be characterized by p hidden topics such that $\mathbf{V}$ can be factorized into the multiplication of two hidden factor Matrices $\mathbf{W}_{d\times p}$ and $\mathbf{H}_{p\times w}$, i.e., $\mathbf{V} \approx \mathbf{W} \times \mathbf{H}$. In real-world applications, the number of topics p is chosen between 50 and 200. Typically, d and w are much larger than p. For example, in the Netflix contest dataset, $d = 480,189$ and $w = 17,770$.

There are two updates for Matrix $\mathbf{H}$ and $\mathbf{W}$ during each iteration. The common matrix multiplication of $\mathbf{W} \times \mathbf{H}$ is not shared between the two compute steps (in Lines 7 and 8), because $\mathbf{H}$ is updated during the execution. Observe that the matrix chain multiplications $\mathbf{W}^T \times \mathbf{W} \times \mathbf{H}$ and $\mathbf{W} \times \mathbf{H} \times \mathbf{H}^T$ involve more than one matrix multiplication. The order of execution on multiple matrix multiplications should be chosen carefully to avoid generating intermediate matrices of large sizes. The matrix metadata records several properties, e.g., the dimension, the sparsity (dense or sparse), and the storage format. From the metadata of the input matrices, it should be possible to infer the dimensions of intermediate matrices.

For computing $\mathbf{W} \times \mathbf{H} \times \mathbf{H}^T$, there are two possible execution orders, i.e., computing $\mathbf{W} \times \mathbf{H}$ first and producing an intermediate result with $480,189 \times 200 \times 17,770$ arithmetic multiplications; or computing $\mathbf{H} \times \mathbf{H}^T$ first and producing an intermediate result with $200 \times 17,770 \times 200$ arithmetic multiplications; assuming the Netflix contest dataset when p is set to 200. The two execution plans differ greatly in the dimensions of the intermediate matrices, and result in different computation costs. The plan generation becomes even more intricate when sparse matrices are involved. It is usually difficult to obtain accurate estimates on the sparsity of the computed intermediate matrices, which is directly related to the computation cost. Another common feature of matrix programs is the mix between element-wise operations and matrix-matrix multiplications. An eager plan generator that arranges a sequential execution incurs unnecessary memory overhead for intermediate matrices. An optimizer for matrix computation is needed to leverage features of matrix programs that reduce computation and memory overhead.

In a distributed setup, the communication overhead may become a bottleneck in matrix computations. Load-balanced data partitioning schemes, e.g., hash-based schemes, where a hash function distributes rows evenly across the partitions, may not be efficient for matrix operations with data dependencies. Data dependencies between compute steps in a matrix program are prevalent, e.g., an update of Matrix $\mathbf{H}$ (Line 7) is fed to compute a new update of Matrix $\mathbf{W}$ (Line 8). The matrix multiplication $\mathbf{H} \times \mathbf{H}^T$ will requires re-partitioning if a hash-based scheme is used for $\mathbf{H}$. Thus, optimizing the data partitioning of input and intermediate matrices is also a critical step for efficiently executing matrix programs.

In this chapter, we introduce MatFast, an in-memory distributed matrix computation processing system. MatFast has (1) a matrix program optimizer to identify and leverage special features of the input matrices to reduce computation cost and memory footprint, and (2) a matrix data partitioner to mitigate communication overhead. The matrix program optimizer uses a cost model and heuristic rules to dynamically generate an execution plan. MatFast uses a second cost model to par-

tition the input and intermediate matrices to minimize communication overhead. To improve compute performance on local matrices, MatFast leverages a block-based strategy for efficient local matrix computations. MatFast is designed as a Spark library that uses Spark's standard dataflow operators.

The main contributions of this chapter are as follows:

- We develop MatFast, a matrix computation system for efficiently processing and optimizing matrix programs in a distributed in-memory environment.
- We introduce a cost model to accurately estimate the sparsity of sparse matrix multiplications, and propose heuristic rules to rewrite special features of a matrix program for mitigating memory footprint.
- We introduce a second cost model to distribute the matrix data partitions among various workers for a communication-efficient execution.
- We evaluate MatFast against state-of-the-art distributed matrix computation systems using real and synthetic datasets. Experimental results illustrate up to an order of magnitude enhancement in performance.

The rest of this chapter proceeds as follows.

Section 3.2 gives an overview of MatFast, and its major components. Section 3.3 presents the plan generation strategies of MatFast. Especially, static heuristic rule-based optimizations and dynamic cost-based optimizations are introduced. Section 3.4 discusses the detailed implementation of the system and its design decisions. Section 3.5 provides the related work. Section 3.6 presents the datasets and experimental evaluation of the system via a number of ML applications. Finally, Section 3.7 contains conclusion remarks.

3.2 An Overview of Distributed Processing of Matrix Computations

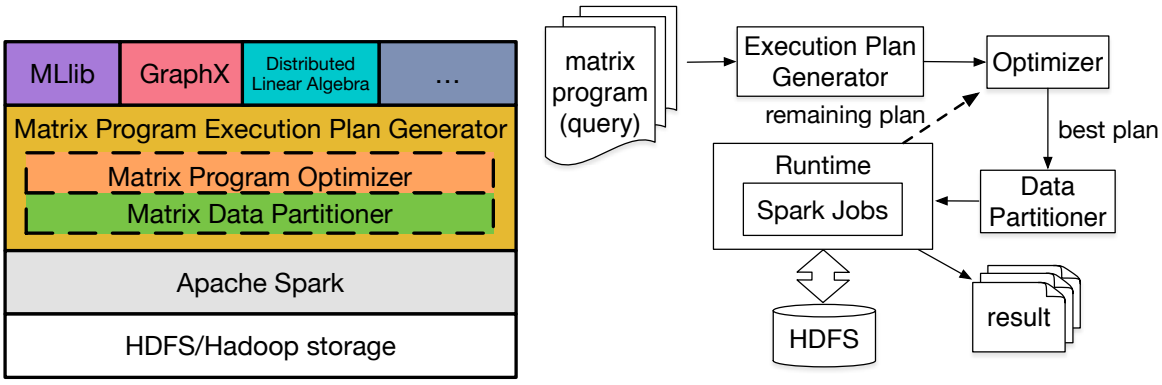
To facilitate matrix computation, we realize an execution plan generator for evaluating matrix expressions over in-memory matrix data. Given an analytic task, e.g.,

an ML algorithm, that involves multiple matrix expressions, these expressions are extracted and are optimized to generate a compute- and memory-efficient logical evaluation plan. Then, we develop a cost model to decide on how the input and intermediate matrix data are partitioned based on data dependencies. Finally, each worker adopts a block-based matrix storage to execute computations locally.

Refer to Figure 3.1 for illustration. MatFast consists of three major components: a plan generator for executing matrix programs, a query optimizer, and a data partitioner. These components leverage rules to transform a matrix expression (that is extracted from a high-level application) to an optimized execution plan in a distributed environment. Figure 3.1b gives the workflow among the various components. For each matrix expression or query (a matrix expression is a query for MatFast), the execution plan generator produces an initial query evaluation plan tree that is pipelined into the query optimizer to apply cost-based dynamic analysis and rule-based rewriting heuristics. The matrix data partitioner assigns partitioning schemes to input and intermediate matrices based on a cost model. For matrix expressions that involve sparse matrix multiplications, a globally optimal execution plan cannot be determined by a single pass on the plan tree due to inaccurate estimates on the computation cost of the intermediate matrices. MatFast adopts a greedy approach to progressively generate an execution order for sparse matrix chain multiplications. The dashed arrow in Figure 3.1b refers to the dynamic optimizations of these cases.

3.3 Plan Generation for Efficient Query Execution

We present how to generate a computation- and communication-efficient execution plan for a matrix expression. First, we present a sampling-based technique to estimate the computation cost for sparse matrix chain multiplications in a single statement. Next, we introduce rule-based heuristics to identify special features of a matrix expression for memory efficiency. Finally, we present a cost model to estimate



(a) Components of MatFast (b) Architecture of MatFast

Figure 3.1.: An overview of MatFast.

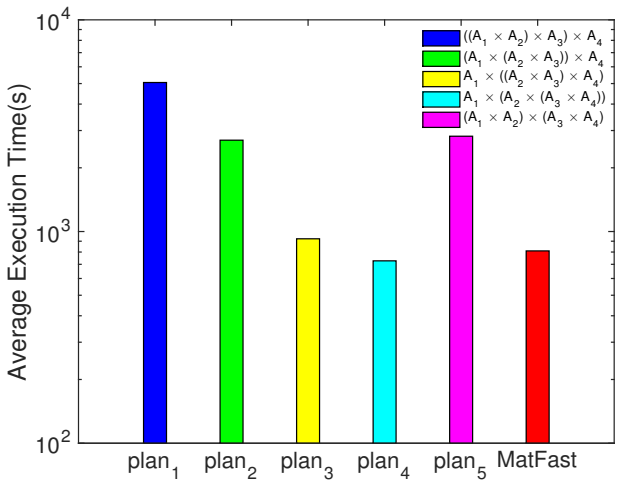


Figure 3.2.: Computation costs of different plans.

the communication overhead for optimizing the data partitioning of individual input and intermediate matrices in the matrix expression.

3.3.1 Cost-based Dynamic Optimization

Matrix chain multiplication is commonly found in random walk [17] and matrix factorization [14] applications. We distinguish between dense and sparse matrix

chain multiplications. For dense matrix chain multiplications, MatFast exploits the classical dynamic programming approach [42] to determine the optimal order of the matrix pair multiplications. The cost of multiplying two dense matrices is defined as the number of arithmetic floating point multiplications. The computation cost of $\mathbf{A}_{m\times q} \times \mathbf{B}_{q\times n}$ can be estimated by mqn floating point multiplications. However, for sparse matrix chain multiplications, we cannot apply the same dynamic programming approach, because the computation cost of intermediate product matrices would be over-estimated. This cost depends not only on the dimensions of the input matrices but also on several other factors, e.g., matrix sparsity and the locations of non-zero entries. Figure 3.2 gives the average runtime of various plans for a sparse matrix chain multiplication of length 4. The cost varies significantly between the best and the worst plan. To better estimate this computation cost, various sparsity estimation methods, e.g., average-case estimation [43] and worst-case estimation [41, 44], can be used and are explained below.

Given a sparse matrix, the associated metadata contains the dimension and sparsity information, i.e., the number of rows m, columns n, and the sparsity ρ, where $\rho = N_{nz}/(mn)$, N_{nz} is the number of non-zero entries. For matrix-matrix multiplication $\mathbf{C}_{m\times n} = \mathbf{A}_{m\times q} \times \mathbf{B}_{q\times n}$, estimating the sparsity ρ_{C} of Matrix $\mathbf{C}$ is difficult, and is usually interpreted as the probability of non-zero entries in a matrix, under the uniform distribution assumption. Thus, the average- and worst-case estimations predict sparsity as $\rho_{\text{C}} = 1 - (1 - \rho_{\text{A}}\rho_{\text{B}})^q$, and $\rho_{\text{C}} = \min(1, \rho_{\text{A}}q) \times \min(1, \rho_{\text{B}}q)$, respectively.

For matrices derived from real-world applications, e.g., online social networks, citation networks, protein-protein interactive networks, non-zero entries usually follow non-uniform distributions. Average-case estimation works poorly for these matrices. The node degrees follow a power law distribution [45], where certain rows and columns contain substantially more non-zero entries than others. Worst-case estimation is pessimistic, and leaves little opportunity for optimization, i.e., it generates a sequential execution plan of the multiplication chain since the sparsity is estimated to 1 when

$\rho_{\text{A}} q \geq 1$ and $\rho_{\text{B}} q \geq 1$. Average- and worst-case estimations are static because they predict sparsity without touching the underlying matrices.

Thus, cost estimation for sparse matrix chain multiplications should conservatively consider data skew. Matrix-matrix multiplication $\mathbf{A} \times \mathbf{B}$ can be interpreted as the summation of the vector outer products between corresponding columns from $\mathbf{A}$ and rows from $\mathbf{B}$, i.e.,

$$\mathbf{A} \times \mathbf{B} = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_k \end{bmatrix} \times \begin{bmatrix} \mathbf{b}_1^T \\ \mathbf{b}_2^T \\ \vdots \\ \mathbf{b}_k^T \end{bmatrix} = \sum_{i=1}^{k} \mathbf{a}_i \mathbf{b}_i^T.$$

This matrix multiplication rule also works for block partitioned matrices, where $\mathbf{A}$ and $\mathbf{B}$ are partitioned into compatible blocks, i.e., the number of columns in block $\mathbf{A}_{ik}$ equals the number of rows in block $\mathbf{B}_{kj}$. The outer-product perspective provides a different way to estimate the cost of sparse matrix multiplications. Intuitively, a larger product value of $\texttt{nnz}(\mathbf{a}_i) \times \texttt{nnz}(\mathbf{b}_i^T)$ leads to a denser multiplication matrix, where $\texttt{nnz}(\mathbf{A})$ denotes the number of non-zero entries in Matrix $\mathbf{A}$. However, it is unaffordable to calculate each $\texttt{nnz}(\mathbf{a}_i) \times \texttt{nnz}(\mathbf{b}_i^T)$ for large matrices with millions of rows or columns. The optimizer needs a sketch about the exact cost.

To obtain an accurate cost estimation of sparse matrix chain multiplications, MatFast adopts a sampling-based approach to sketch the positions of the non-zero entries. A good sampling method needs to capture the densest columns and rows. If the number of non-zero entries in a row (column) of a sparse matrix follows a power law distribution, and the rows (columns) are in the descending order with respect to the number of nonzero entries, then it is ideal to select the first few rows and columns for estimating the computation cost of multiplying the matrices. If no prior knowledge is available for the input, MatFast adopts a simple random-sampling method, e.g., systematic sampling [46], to estimate the computation cost of the multiplication. This

cost estimation can be generalized to block partitioned sparse matrix multiplication as follows,

$$C_{comp}(\mathbf{A} \times \mathbf{B}) = \max_{k \in \mathcal{S}} \{c_k * r_k\},$$
$$\text{where } c_k = \sum_i \texttt{nnz}(\mathbf{A}_{ik}), r_k = \sum_j \texttt{nnz}(\mathbf{B}_{kj}),$$

where $C_{comp}(\mathbf{X})$ denotes the computation cost of calculating Matrix $\mathbf{X}$, and $\mathcal{S}$ is the set of the sampled column (row) block indices, and c_k (r_k) is the number of non-zero entries in the k-th column (row) block. The maximum operator is used because a larger value of $c_k * r_k$ indicates a denser product matrix.

Analysis of Cost Estimation with Sampling.

If the distribution of the non-zero entries is provided by a user, MatFast samples rows (columns) according to the distribution. If MatFast samples input matrices with a random sampling method, the probability of accurately estimating the cost can be modeled as follows. Suppose there are n columns in Matrix $\mathbf{A}$, n rows in Matrix $\mathbf{B}$, and w column-row pairs, whose products achieve the maximum product. The probability that the maximum product is chosen in s samples is given by,

$$P = 1 - \binom{n-w}{s}\binom{n}{s}^{-1}.$$

By sampling s row-column pairs, there are totally $\binom{n}{s}$ possible combinations. The chance that the maximum pair is not chosen among w pairs is $\binom{n-w}{s}\binom{n}{s}^{-1}$. Thus, P can be computed by the complementary event. Similarly, for a block partitioned matrix, this probability is as follows:

$$\hat{P} = 1 - \binom{\hat{n}-\hat{w}}{s}\binom{\hat{n}}{s}^{-1},$$

where $\hat{n}$ is the number of column (row) blocks of matrix $\mathbf{A}$ ($\mathbf{B}$), $\hat{n} = n/\ell$ (ℓ is the block size), $\hat{w}$ is the number of blocks that achieves the maximum product value. In practice, $\hat{P} \geq P$ and the probability of accurate cost estimation is improved.

Running Example.

Given a sparse matrix chain multiplication $\mathbf{A}_1 \times \mathbf{A}_2 \times \mathbf{A}_3 \times \mathbf{A}_4$, MatFast dynamically generates an execution plan. Initially, the sampling index set $\mathcal{S}$ is determined

by a random sampling method. The number of non-zero entries are collected for the sampled rows and columns. Next, the costs for pair-wise adjacent matrix multiplications are computed, i.e., $c_1 = C_{comp}(\mathbf{A}_1 \times \mathbf{A}_2)$, and similarly for c_2 and c_3. Say, c_2 is the cheapest. The multiplication chain is computed a step further and is reduced to $\mathbf{A}_1 \times \mathbf{M}_{23} \times \mathbf{A}_4$, where $\mathbf{M}_{ij}$ is the intermediate product matrix of $\mathbf{A}_i$ and $\mathbf{A}_j$. Then, the sampling is conducted again on $\mathbf{M}_{ij}$. Notice that the sampled statistics can be reused for the existing matrices, e.g., $\mathbf{A}_1$ and $\mathbf{A}_4$. The sampling-based cost estimation repeats until the chain is reduced to a single matrix. We identify the following features that occur frequently in matrix programs:

3.3.2 Rule-based heuristics

Matrix expressions have various features that may induce heavy memory footprints. We identify the following features that occur frequently in matrix programs: (1) low-rank matrix updates, (2) chains of multiple element-wise matrix operators, and (3) loop structures that reflect iterative executions. MatFast handles these features by using heuristics to generate a memory-efficient execution plan.

Identifying and Preserving Low-rank Matrix Updates.
Low-rank matrix updates are widely used in ML models due to the popularity of latent variables [47]. Latent variables are utilized in many disciplines, e.g., economics, machine learning, bioinformatics, natural language processing, and social sciences. For example, the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm [48] and its variant limited memory BFGS (ℓ-BFGS) are widely used quasi-Newton methods for solving unconstrained nonlinear optimization problems. Low-rank matrix update is a critical step of BFGS and is stated as follows:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{\mathbf{y}_k \times \mathbf{y}_k^T}{\mathbf{y}_k^T \times \mathbf{s}_k} - \frac{\mathbf{B}_k \times \mathbf{s}_k \times \mathbf{s}_k^T \times \mathbf{B}_k}{\mathbf{s}_k^T \times \mathbf{B}_k \times \mathbf{s}_k},$$

where $\mathbf{B}_k$ is the approximate Hessian matrix, $\mathbf{s}_k$ is the line search step, and $\mathbf{y}_k$ is the difference of the gradient. $\mathbf{y}_k^T \times \mathbf{s}_k$ and $\mathbf{s}_k^T \times \mathbf{B}_k \times \mathbf{s}_k$ are two scalars that can be easily computed and shared among workers. Notice that there are two low-rank (rank-1)

matrix updates in each iteration, i.e., $\mathbf{y}_k \times \mathbf{y}_k^T$ and $\mathbf{B}_k \times \mathbf{s}_k \times \mathbf{s}_k^T \times \mathbf{B}_k$. An ignorant query optimizer generates a sequential execution plan for each intermediate matrix, i.e., both low-rank update matrices have to be explicitly computed and materialized during the execution. However, multiplying low-rank matrices usually produces dense matrices of very high dimensions, and incurs heavy memory overhead.

Given a matrix expression, MatFast analyzes the dimensions of the input matrices, and identifies low-rank matrix updates. It defers the evaluation of low-rank matrix updates to reduce the memory footprint. Low-rank matrix updates usually involve matrix components of low dimensions. Thus, storing and transmitting the low-rank matrix components are more efficient than materializing the matrix product. For example, matrix $\mathbf{y}_k$ and $\mathbf{B}_k \times \mathbf{s}_k$ are two vectors but their corresponding rank-1 updates $\mathbf{y}_k \times \mathbf{y}_k^T$ and $\mathbf{B}_k \times \mathbf{s}_k \times \mathbf{s}_k^T \times \mathbf{B}_k$ are dense. Thus, deferring the computation of low-rank matrix multiplications can reduce communication overhead. To evaluate the updated Hessian matrix $\mathbf{B}_{k+1}$, first, we broadcast vector $\mathbf{y}_k$ and $\mathbf{B}_k \times \mathbf{s}_k$ to all workers. Then, each element of the low-rank matrix multiplication is computed from the vectors on the fly without storing the matrix product explicitly.

Folding of Matrix Operators.

Matrix element-wise operations are prevalent in ML algorithms. For GNMF (Code 1.1), the algorithm updates Matrix $\mathbf{W}$ with element-wise multiplications and divisions, i.e., $\mathbf{W} = \mathbf{W} * (\mathbf{V} \times \mathbf{H}^T)/(\mathbf{W} \times \mathbf{H} \times \mathbf{H}^T)$. These expressions are generalized as $\mathbf{A}_1 \star \mathbf{A}_2 \star \cdots \star \mathbf{A}_k$, where all $\mathbf{A}_i$'s have the same dimension and $\star$'s are element-wise operators, i.e., $\star \in \{+, *, /\}$. For the update of $\mathbf{W}$, $\mathbf{A}_1 = \mathbf{W}, \mathbf{A}_2 = \mathbf{V} \times \mathbf{H}^T$, and $\mathbf{A}_3 = \mathbf{W} \times \mathbf{H} \times \mathbf{H}^T$. To generate an execution plan for this expression, a naive query optimizer generates a "*left-deep tree*" plan. The left part of Figure 3.3 shows the left-deep tree plan for updating Matrix $\mathbf{W}$, where the dashed rectangles represent the subtrees for $\mathbf{A}_2$ and $\mathbf{A}_3$. The problem is that the inner nodes of the left-deep tree plan must be materialized, e.g., $\mathbf{W} * (\mathbf{V} \times \mathbf{H}^T)$, before the element-wise division. A left-deep tree plan induces multiple compute steps and memory overhead for storing the intermediate matrices.

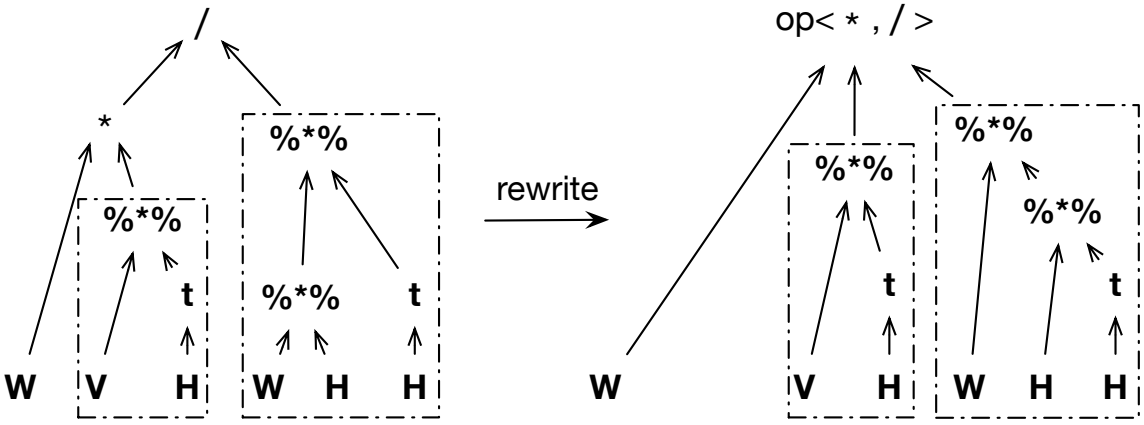


Figure 3.3.: Element-wise operators folding.

Given a matrix expression, MatFast identifies element-wise matrix operations and then organizes them in a "*bushy tree*" execution plan. A bushy tree plan benefits from circumventing materializing the intermediate matrices. The right part of Figure 3.3 illustrates the bushy tree plan for updating Matrix $\mathbf{W}$. To facilitate low-level executions, MatFast's optimizer generates a tree node of a new compound operator in the form "`op<binop_list>`". The compound operator records each element-wise operator and the corresponding input matrices. For example, "`op<*,/>`" in Figure 3.3 encodes the chain of matrix element-wise multiplication and division operations among $\mathbf{W}$, $\mathbf{A}_2$, and $\mathbf{A}_3$. Notice that the dashed rectangle also gives an optimized plan for intermediate Matrix $\mathbf{A}_3$. The optimization reflects the execution plan of $\mathbf{W} \times \mathbf{H} \times \mathbf{H}^T$.

Eliminating Common and Invariant Expressions.
Many ML models and matrix computations are iterative in nature. Accelerating loop executions provides performance improvement for matrix programs. To optimize loop executions, MatFast identifies and eliminates recurring common subexpressions (CSE) [49]. It recognizes loop-constant subexpressions, moves them out of the loop, and saves them for reuse. Thus, redundant computation and memory footprint are mitigated.

For example, the key update step of PageRank (Code 1.2) is a matrix-vector multiplication,

$$\mathbf{x}_{k+1} = \alpha \mathbf{P}\mathbf{x}_k + (1-\alpha)\mathbf{v},$$

where α is a constant scalar, $\mathbf{P}$ is the stochastic link matrix, and $\mathbf{v}$ is a constant restart vector. Notice that the constant expressions $\alpha\mathbf{P}$ and $(1-\alpha)\mathbf{v}$ are evaluated repetitively during iterations. Therefore, we move the constant expressions out of the iteration loop, and the redundant computations are avoided.

3.3.3 Generation of Execution Plans Involving Big Matrix-data Partitioning

When an optimized plan is distributed among a set of workers, its execution may suffer from heavy communication overhead due to inconsistent data partitioning schemes between dependent matrices. For a typical pipeline of matrix operators, the output of an operator always serves as an input to another. For GNMF (Code 1.1), the computation of $\mathbf{H} \times \mathbf{H}^T$ relies on the previous step. If $\mathbf{H}$ is partitioned in the default hash-based scheme, then we need to repartition $\mathbf{H}$ before the actual execution, and hence inducing extra communication cost. Thus, choosing a consistent matrix partitioning scheme for an intermediate matrix is essential to reduce communication overhead. In this section, we introduce MatFast's matrix data partitioning schemes. Then, we present a cost model to efficiently partition matrix data.

Partitioning Schemes.

MatFast supports the following three widely used matrix data partitioning schemes: *Row* ("*r*"), *Column* ("*c*"), and *Block-Cyclic* ("*b-c*"). Moreover, the *Broadcast* ("*b*") scheme is supported for sharing a matrix of low dimensions or a single vector. The Row and Column schemes place all the elements in the same row and column on a worker, respectively. The Block-Cyclic scheme partitions a matrix into many more blocks than the number of available workers, and assigns blocks to workers in a round-robin manner so that each worker receives several non-adjacent blocks. Different schemes introduce different communication costs for various matrix operators.

They should be assigned to input matrices on their own merits. For example, for matrix-matrix multiplications with cross product plans [36], the Block-Cyclic scheme incurs extra shuffle cost to aggregate rows (columns) together. Row and column schemes assign related matrix elements on the same executor for matrix-matrix multiplication with no extra aggregation costs. Therefore, we introduce a cost model to evaluate the communication costs of different partitioning schemes for the various matrix operators.

Cost Model for Communication.

The communication cost incurred by a matrix expression can be modeled by,

$$C_{comm}(expr_i) = \sum_{j=1}^{m} C_{comm}(op_j, s_{j_{i1}}[, s_{j_{i2}}], s_{j_o}),$$

where matrix expression $expr_i$ contains j matrix operators, and each operator takes some inputs in schemes $s_{j_{i1}}[, s_{j_{i2}}]$, and produces an output in scheme s_{j_o}. $[, s_{j_{i2}}]$ represents an optional argument as we support both unary and binary operators. For the unary operator (i.e., matrix transpose) op, $C_{comm}(op, s_i, s_o)$ is characterized by the input matrix and partitioning schemes in Table 3.1, where $\mathbf{A}$ is the input matrix, s_i and s_o are the partitioning schemes of the input and the output, respectively, and N is the number of the workers in the cluster. $|\mathbf{A}|$ refers to the size of Matrix $\mathbf{A}$, i.e., $|\mathbf{A}| = mn$ if $\mathbf{A}$ is an m-by-n dense matrix; and it means $\texttt{nnz}(\mathbf{A})$ if $\mathbf{A}$ is sparse. If the input matrix is partitioned in the Row scheme, then the transposed matrix is naturally partitioned in the Column scheme. Therefore, no communication cost is introduced. However, if the input matrix is partitioned in the Row scheme and the output matrix is also required to be partitioned in the Row scheme, then the matrix data must be shuffled to satisfy the requirement. This results in shuffling the whole matrix.

Similarly, for matrix-scalar operators (e.g., multiplying a matrix by a constant), Table 3.2 gives the communication costs for the various schemes. If the input and the output are partitioned in the same scheme, then there is no communication. Notice

Table 3.1.: Communication cost of matrix transpose

$s_o \setminus s_i$	r	c	b	b-c
r	$\|\mathbf{A}\|$	0	0	$\|\mathbf{A}\|$
c	0	$\|\mathbf{A}\|$	0	$\|\mathbf{A}\|$
b	$N\|\mathbf{A}\|$	$N\|\mathbf{A}\|$	0	$N\|\mathbf{A}\|$
b-c	$\|\mathbf{A}\|$	$\|\mathbf{A}\|$	0	$\|\mathbf{A}\|$

Table 3.2.: Communication cost of matrix-scalar operators

$s_o \setminus s_i$	r	c	b	b-c
r	0	$\|\mathbf{A}\|$	0	$\|\mathbf{A}\|$
c	$\|\mathbf{A}\|$	0	0	$\|\mathbf{A}\|$
b	$N\|\mathbf{A}\|$	$N\|\mathbf{A}\|$	0	$N\|\mathbf{A}\|$
b-c	$\|\mathbf{A}\|$	$\|\mathbf{A}\|$	0	0

that no communication is incurred if the inputs are partitioned in the Broadcast scheme.

Let $C_{comm}(op, s_{i1}, s_{i2}, s_o)$ be the cost function for a matrix element-wise operator that is illustrated in Table 3.3. From the table, the matrix element-wise operators introduce no communication overhead if both input matrices are partitioned (1) in the same scheme as the output, (2) at least one of the inputs is partitioned in the Broadcast scheme and the other one has the same scheme as the output.

Matrix-matrix multiplication is a bit more complicated. We do not use the Block-Cyclic scheme as it incurs more overhead than the other schemes. The cost function is given in Table 3.4. Notice that the cells with 0's in the table indicate no cost, e.g., when the inputs are partitioned in the Row scheme, the Broadcast scheme, and the output is in the Row scheme.

Table 3.3.: Communication cost of element-wise operators

s_o	(s_{i1}, s_{i2}) Communication Cost		
r	{(r, r), (r, b), (b, r), (b, b)} 0		else $\|\mathbf{A}\|$
c	{(c, c), (c, b), (b, c), (b, b)} 0		else $\|\mathbf{A}\|$
b	(b, b) 0	$s_{i1} = s_{i2}$ or {(b, *), (*, b)} $N\|\mathbf{A}\|$	else $(N+1)\|\mathbf{A}\|$
b-c	{(b-c, b-c), (b, b), (b-c, b), (b, b-c)} 0	{(b-c, *), (*, b-c)} $\|\mathbf{A}\|$	else $2\|\mathbf{A}\|$

With the cost functions introduced above, MatFast assigns the partitioning schemes having minimum costs to the input and intermediate matrices, i.e.,

$$s_{i1(i2)} \leftarrow \arg\min_{s_{i1(i2)}} C_{comm}(op, s_{i1}[, s_{i2}], s_o).$$

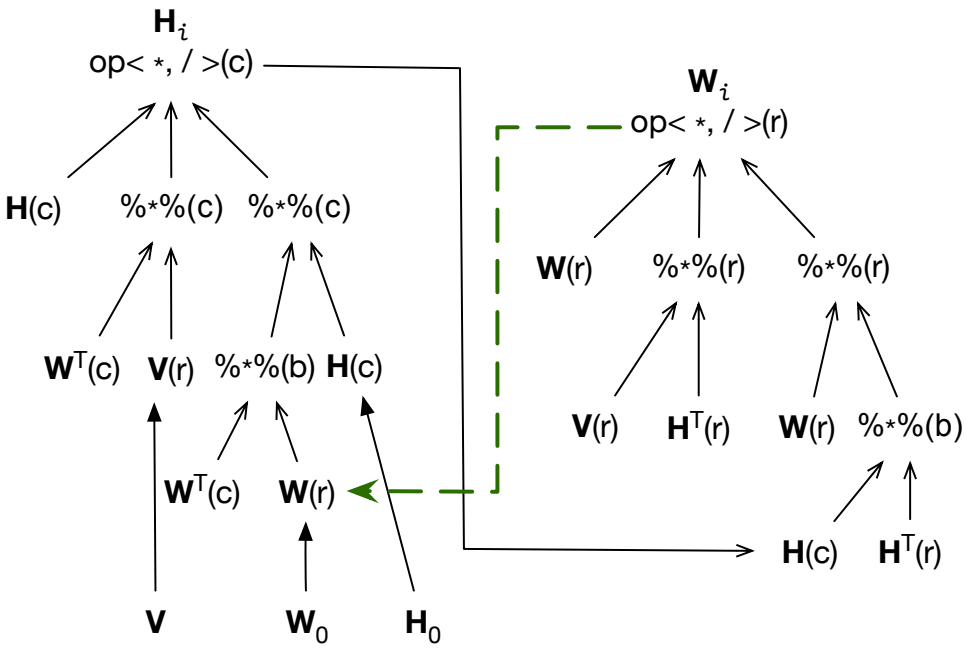
MatFast optimizes the communication cost for a single operator, and assigns the associated scheme to the input. For a matrix expression consisting of several operators, the entire expression is greedily optimized by tuning each operator.

Table 3.4.: Communication cost of matrix-matrix multiplications. $C(s_{i1}, s_{i2})$ is the cost when the 2 matrices are partitioned in schemes s_{i1} and s_{i2}.

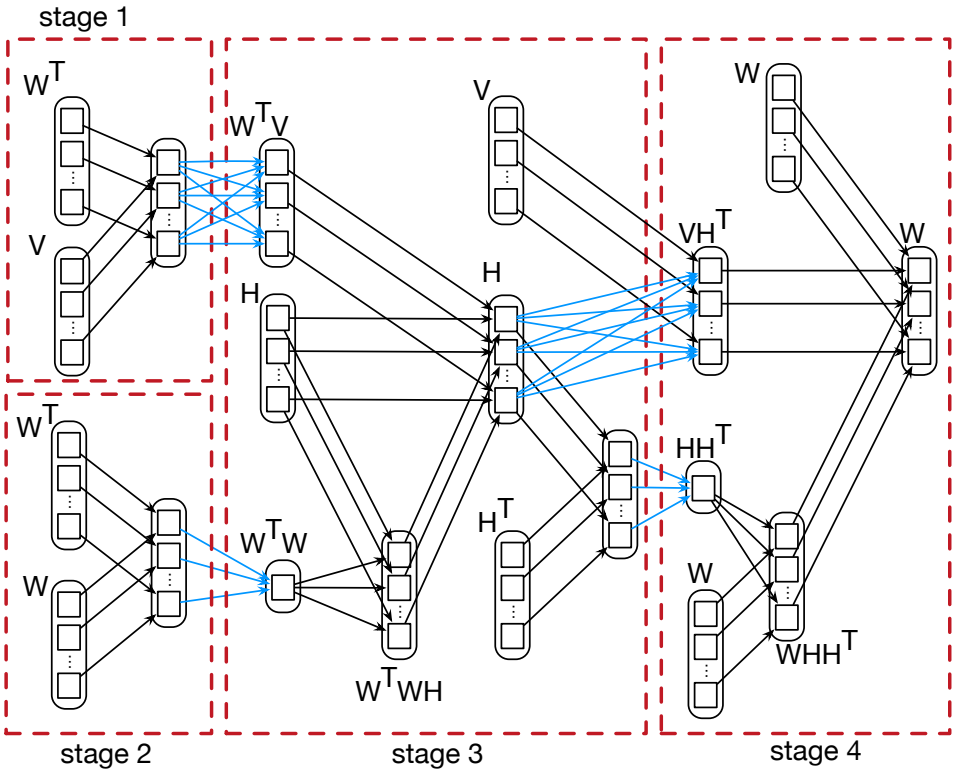
	(s_{i1}, s_{i2})								
s_o	(r, r)	(r, c)	(r, b)	(c, r)	(c, c)	(c, b)	(b, r)	(b, c)	(b, b)
r	$N\|\mathbf{B}\|$	$N\|\mathbf{B}\|$	0	$N\|\mathbf{AB}\|$	$N(\|\mathbf{A}\| + \|\mathbf{B}\|)$	$\|\mathbf{A}\|$	$N\|\mathbf{AB}\|$	$\|\mathbf{AB}\|$	0
c	$N(\|\mathbf{A}\| + \|\mathbf{B}\|)$	$N\|\mathbf{A}\|$	$\|\mathbf{AB}\|$	$N\|\mathbf{AB}\|$	$N\|\mathbf{A}\|$	$N\|\mathbf{AB}\|$	$\|\mathbf{B}\|$	0	0
b	$\min\{N\|\mathbf{A}\| + C(b, r),$ $N\|\mathbf{B}\| + C(r, b)\}$	$N\min\{\|\mathbf{A}\|, \|\mathbf{B}\|\}$ $+N\|\mathbf{AB}\|$	$N\|\mathbf{AB}\|$	$2(N-1)\|\mathbf{AB}\|$	$\min\{N\|\mathbf{A}\| + C(b, c),$ $N\|\mathbf{B}\| + C(c, b)\}$	$\min\{\|\mathbf{A}\| + N\|\mathbf{AB}\|,$ $2(N-1)\|\mathbf{AB}\|\}$	$\min\{\|\mathbf{B}\| + N\|\mathbf{AB}\|,$ $2(N-1)\|\mathbf{AB}\|\}$	$N\|\mathbf{AB}\|$	0

Algorithm for Plan Generation and Partitioning Scheme Assignment. Algorithm 1 describes plan generation in MatFast. The input is a matrix program P, and the output is an optimized execution plan tree T with an optimized partitioning scheme at each node. MatFast applies the rule-based heuristics to each expression of P. If the expression contains no matrix operator, then the variable is parsed and associated with the corresponding metadata (Line 28), e.g., loading matrix data or storing results to HDFS. If an expression contains matrix chain multiplications, the execution plan is determined by the matrix types (Lines 17-24), i.e., a classical dynamic programming approach is invoked for dense matrix chains; otherwise, dynamic cost-estimation is triggered for sparse matrix chain multiplications (Line 22). Then, the optimized expression is inserted into the plan tree T (Line 25). Finally, Procedure assignParitionScheme (Algorithm 2) assigns partitioning schemes to matrices based on the cost model. The scheme assignment starts from the root of the plan tree T. For the root node, the scheme is determined by the nature of the output, i.e., recurring in a loop, participating in matrix-matrix multiplications, or only involving in element-wise operations. For an internal node, the Broadcast scheme is assigned if it consists of low-rank matrix updates. Otherwise, an internal node is assigned with a scheme such that it introduces minimum communication cost (Lines 17 and 20). After scheme assignment, the execution is organized into several *stages*, where the operations are packed together such that no communication is introduced in the same stage.

GNMF Running Example. The optimized execution plan tree is given in Figure 3.4a. Procedure AssignPartitionScheme traverses from the root node to all leaf nodes, and assigns a partitioning scheme to each input and intermediate matrix. The left and right subtrees are the execution plans for updating Matrices $\mathbf{H}$, and $\mathbf{W}$, respectively. The dashed arrows indicate loop execution. The root node $\mathbf{W}_i$ participates in matrix-matrix multiplication and loop execution. The number of rows in $\mathbf{W}_i$ is significantly bigger than that of columns. Thus, the Row scheme leads to a more balanced data distribution. The



(a) Optimized query execution plan of GNMF



(b) Physical execution plan of GNMF

Figure 3.4.: Execution plan with matrix data partitioning scheme of GNMF. In the physical plan, the blue lines denote the data shuffle among different data partitions. The dashed red rectangles denote different stages for the execution on Spark.

cost model for the element-wise operator aids in assigning the Row schemes to $\mathbf{W}_i$'s child nodes. To determine the partitioning schemes for $\mathbf{V}$ and $\mathbf{H}^T$, the procedure checks the cost model table for matrix-matrix multiplication. The dimension of $\mathbf{V}$ is larger than that of the product $\mathbf{V} \times \mathbf{H}^T$. Thus, partitioning both matrices in the Row scheme is cheaper than other strategies, i.e., $17,770 \times 200 \times N$. Matrix $\mathbf{H} \times \mathbf{H}^T$ is partitioned in the Broadcast scheme due to its tiny dimension (200×200). Similarly, $\mathbf{H}$ uses the Column scheme and $\mathbf{H}^T$ uses the Row scheme with no cost. $\mathbf{H}_i$'s subtree is processed similarly. Figure 3.4b gives the physical execution in the cluster. The dashed boxes indicate the different execution stages.

3.4 Local Execution and System Implementation

Once the query execution with matrix partitioning schemes is generated, each compute node locally performs matrix computations. We use block matrices as a basic unit for manipulation to store matrix data in the distributed memory. We discuss briefly the system implementation on top of Apache Spark. In this section, we will describe the matrix data partitioner layer and how the optimizer determines the partitioning schemes for input and intermediate matrices.

3.4.1 Physical Storage of a Local Matrix

To better utilize spatial locality of nearby entries in a matrix, we use block matrices to store matrix data in the distributed memory. A matrix block is the basic unit for storage and computation. Figure 3.5 illustrates that Matrix $\mathbf{A}$ is partitioned into blocks of size 3×3, where each block is stored as a local matrix. To fully exploit the parallelism of the workers, block size is an important parameter for efficient execution. A large block size leads to computation skew, while a small block size results in heavy communication costs. For the sake of simplicity, we only consider square blocks. Figure 3.5 illustrates an example storage layout of a block matrix. The block size may not be applicable to the last row (column) block, e.g., the row block

Algorithm 1: Execution plan generation

Input: Matrix program P

Output: Execution plan tree T

```
1  T ← ∅
2  L ← P.getExprList() // expressions in sequential order
3  for expr in L do
4      currExpr ← expr
5      if expr.containsMatrixOperator() then
6          // low-rank matrix preservation
7          if expr.containsLowRankMatrix() then
8              currExpr ← preserveLowRank(currExpr)
9          end
10         if expr.elementOperator() > 1 then
11             currExpr ← createCompound(currExpr) // operator folding
12         end
13         // loop invariant extraction
14         if expr.isLoop() && expr.containsConst() then
15             currExpr ← extractConst(currExpr)
16         end
17         if expr.containsMatrixChainMult() then
18             if isDenseChain(expr) then
19                 currExpr ← densePlan(currExpr)
20             end
21             else
22                 currExpr ← sparsePlan(currExpr)
23             end
24         end
25         T.add(currExpr)
26     end
27     else
28         load data into memory and extract metadata
29     end
30 end
31 assignPartitionScheme(T, null)
32 return T
```

Algorithm 2: assignPartitionScheme(T, q)

```
// Plan tree T, output matrix partitioning scheme q
// for root node
if q = null then
    if T.type = 'compound' && ((isInLoop(T) && !inMatMult(T)) ||
     !isInLoop(T)) then
        T.scheme ← 'b-c'
    end
    else
        Choose p ∈ {'r', 'c'} based on the metadata of T, and T.scheme ← p
    end
end
// for an internal node
else if T.scheme = null then
    if isLowRankMult(T) || isTinySize(T) then
        T.scheme ← 'b'
    end
    if op is unary then
        T.scheme ← arg min_{s_i} C_comm(op, s_i, q)
    end
    else
        T.scheme ← arg min_{s_i1 or s_i2} C_comm(op, s_i1, s_i2, q), with respect to the position of T
    end
end
for R in T.children do
    assignPartitionScheme(R, T.scheme)
end
```

with ID = 2. To fully exploit the compute power of CPU cores, MatFast assigns each core 4 matrix blocks to each worker, i.e., $MN = 4WP\ell^2$, where M and N are the dimensions of the matrix, W is the number of workers, P is the number of cores per worker, and ℓ is the block size. To avoid performance degeneration, MatFast limits the smallest block size to be 1000, i.e., $\ell = \max\left\{\sqrt{\frac{MN}{4WP}}, 1000\right\}$.

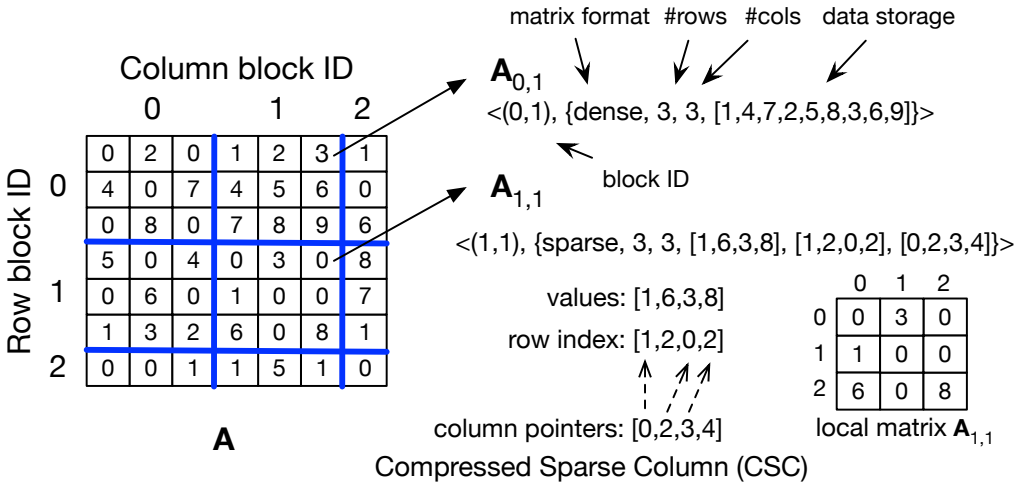


Figure 3.5.: Block matrix storage.

Each local matrix block consists of two components; a block ID and matrix data. A block ID is an ordered pair, i.e., (row-block-ID, column-block-ID). The matrix data field is a quadruple, ⟨matrix format, number of rows, number of columns, data storage⟩. A local matrix block supports dense and sparse matrix storage formats. For the dense format, an array of double precision floating point numbers stores all block entries. For the sparse format, the non-zero entries are stored in Compressed Sparse Column (CSC), and Compressed Sparse Row (CSR) format [50]. The compressed format, say CSC, requires three arrays to store all the data. Array *values* stores all the non-zero entries, and Array *row index* records the row index for the corresponding entry. Array *column pointers* keeps the starting position of each column, and the last entry records the total number of non-zero entries. Figure 3.5 illustrates the dense representation of Block $\mathbf{A}_{0,1}$, and the compressed representation (CSC) of Block $\mathbf{A}_{1,1}$.

An $m \times n$ sparse matrix in CSC format requires $(2N_{nz} + n + 1) \times 8$ bytes of memory, where N_{nz} is the number of nonzero entries.

For dense matrices, we leverage high-performance dense matrix kernels to conduct matrix operations locally, e.g., the LAPACK kernel. Unlike existing systems, e.g., MLlib [40], MatFast operates on sparse matrices in compressed formats directly, without converting to their dense counterparts. Performing local matrix computations in compressed format mitigates the memory footprint for operations among large sparse matrices. This is confirmed by our experiments in the PageRank and sparse matrix chain multiplications case studies.

3.4.2 System Design and Implementation

MatFast is implemented as a library in Spark, and provides Scala API for conducting distributed matrix computations. It uses RDD (Resilient Distributed Datasets) [51] to represent matrix blocks. A driver program orchestrates the executions of the various workers in the cluster. The dimension and sparsity statistics are computed and maintained at the driver program when a matrix is loaded into memory. The optimized execution plan and data partitioning schemes are generated at the driver program as well. The matrix operators are realized via RDD's transformation operations, e.g., `map`, `flatMap`, `zipPartitions`, and `reduceByKey`. Each local matrix has a block ID, and is stored as either a `DenseMatrix` or a `SparseMatrix`. Local matrix operations are optimized by the LAPACK kernel when conducting dense matrix computations. Sparse matrix computations, e.g., multiplication, are conducted in compressed format. The RDD Partitioner class is extended with the four Row, Column, Broadcast, and Block-Cyclic partitioning schemes for distributed matrices. MatFast utilizes the caching mechanism of Spark to buffer a computed matrix when it repeatedly appears in the execution plan. Spark's fault tolerance mechanism applies naturally to MatFast. The Spark cluster is managed by YARN, and a failure in the master node is detected and managed by ZooKeeper. In the

case of master node failure, the lost master node is evicted and a standby node is chosen to recover the master. An open-source version of MatFast is available at `https://github.com/yuyongyang800/SparkDistributedMatrix`.

3.5 Related Work

Matrix computation has been an active research topic for many years in the high-performance computing (HPC) community. Existing libraries, e.g., BLAS [52] and LAPACK [53], provide efficient matrix operators. ScaLAPACK [54] is a distributed variant of LAPACK built on the SPMD (single program, multiple data) model, but it lacks support for sparse matrices, and is prone to machine failures. SpMachO [43] optimizes computation costs for sparse matrix chain multiplications in a single machine setting.

Many systems have been proposed to support efficient matrix computations using Hadoop [30] and Spark [51]. Pegasus [55], a Hadoop-based library, implements a special class of graph algorithms expressed in repeated matrix-vector multiplications. However, optimizing a single operation is restrictive for various matrix applications. Many systems provide interfaces for multiple matrix operations, e.g., HAMA [34], Mahout [35], MLI [39], MadLINQ [56], SystemML [36], and DMac [41]. HAMA and Mahout provide matrix algorithm implementations using the low-level MapReduce APIs that makes it hard to realize new algorithms and tune for performance. MLI is a programming interface built on top of Spark for ML applications. However, MLI does not provide optimizations for sequences of matrix computations.

MadLINQ [56] exploits fine-grained pipelining to explore inter-vertex parallelism. However, it lacks support for efficient sparse matrix operations. SystemML provides an R-like interface for matrix primitives, and translates the script into a series of optimized MapReduce jobs on Hadoop. However, SystemML lacks optimized data partitioning of input and intermediate matrices with respect to the execution. A hybrid parallelization strategy [57] of combining task and data parallelism is proposed

for SystemML to achieve comparable performance to in-memory computations. Column encoding schemes and operations over compressed matrices are proposed for SystemML [58] to handle the case when data does not fit in memory. DMac [41] leverages matrix dependencies to build a communication efficient execution plan, but it does not identify the special features of a matrix program to reduce computation costs and memory footprints. SciDB [59] supports the array data model, but it treats each operator individually without tuning for a series of matrix operators. MatFast differs from these platforms in that (1) MatFast's optimized execution plan is generated progressively by leveraging dynamic sampling-based order selection for sparse matrix chain multiplications and rule-based heuristics for special features of a matrix program, (2) sparse matrix computations are conducted in the compressed formats, and (3) matrix data partitioning scheme assignment based on the optimized plan mitigates communication overhead in the distributed computing environment.

3.6 Performance Evaluation

We study the performance of the optimized execution plans by performing matrix operations from various ML applications. The performance is measured by the average execution time and communication (shuffle) cost. The experiments are conducted on two clusters: (1) a 4-node cluster, where one node has an Intel Xeon(R) E5-2690 CPU, 128GB memory, and a 2TB disk; the other 3, each has an Intel Xeon(R) E5-2640 CPU, 64GB memory, and a 2TB disk. The maximal memory allocation for JVM is 48GB, (2) Hathi cluster[1] has 6 Dell compute nodes. Each has 2 8-core Intel E5-2650v2 CPUs, 32 GB memory, and 48TB of local storage. Spark 1.5.2 runs on YARN with the default configuration.

Case Studies.

We conduct case studies on a series of ML models and matrix computations with

[1] `https://www.rcac.purdue.edu/compute/hathi/`

special features on different datasets. These are PageRank, GNMF, BFGS, sparse matrix chain multiplications, and a biological data analysis.

Datasets.

Our experiments are performed on both real-world and synthetic datasets. The six real-world datasets are: soc-pokec[2], cit-Patents[3], LiveJournal[3], Twitter2010[3], Netflix [15], and 1000 Genomes Project sample[4]. The synthetic datasets are generated by a sparse matrix generator by varying the dimensions, sparsity, skew type, and skew. The skew controls whether non-zero entries distribute in a row- or column-major way. To generate an $m \times n$ matrix with sparsity ρ and skew s in the column-major fashion, the generator produces $mn\rho$ values in a given range. The $mn\rho(1-s)^{j-1}$'s remaining elements are assigned randomly to the j-th column, until all the $mn\rho$ values are assigned. The PageRank experiments on soc-pokec, cit-Patents, LiveJournal, and synthetic sparse matrix chain multiplications are performed on Hathi. The rest are conducted on the first cluster as larger date sets require more memory on each worker node.

Baseline Comparison.

Various matrix computation platforms have been recently proposed on Spark. In particular, we compare MatFast with MLlib [40] and SystemML (`https://github.com/apache/incubator-systemml,0.9`) [36] in Spark batch mode. To validate the proposed optimizations, the results are presented in 2 modes, MatFast and MatFast(opt), where the optimizations for special matrix program features and data partitioning schemes are turned off in MatFast. MatFast partitions matrix data with the default hash partitioner in Spark. Moreover, the open-source library ScaLAPACK [54] and the array-based database SciDB [59] are used for performance evaluation.

[2] `https://snap.stanford.edu/data/`

[3] `http://law.di.unimi.it/webdata/twitter-2010/`

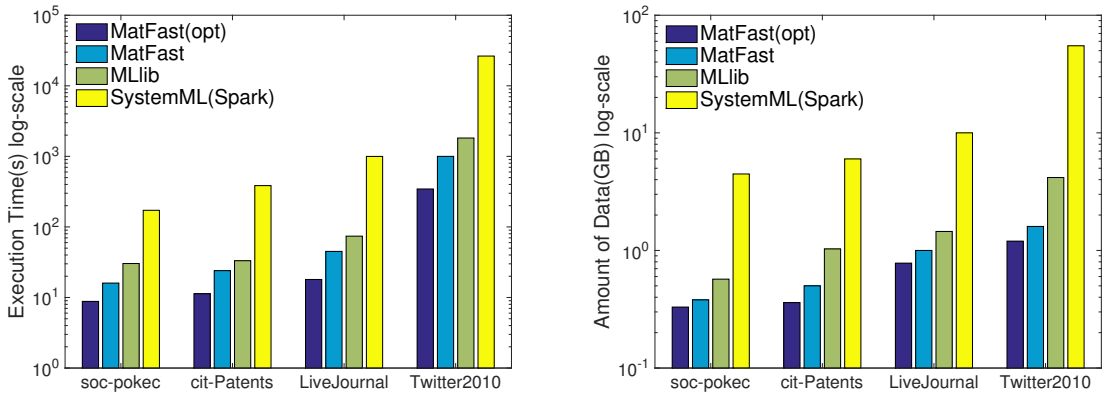
[4] `http://www.1000genomes.org/`

Table 3.5.: Statistics of the social network datasets

Graph	**#nodes**	**#edges**
soc-pokec	1,632,803	30,622,564
cit-Patents	3,774,768	16,518,978
LiveJournal	4,847,571	68,993,773
Twitter2010	41,652,230	1,468,365,182

3.6.1 PageRank

The most expensive compute step of PageRank [17] is updating the vector, i.e., $\mathbf{x}_{k+1} = \alpha\mathbf{P}\mathbf{x}_k + (1-\alpha)\mathbf{v}$, where α is a constant scalar, $\mathbf{P}$ the stochastic link matrix, $\mathbf{v}$ the constant restart vector. The computation can be improved by eliminating constant sub-expressions from the loop, e.g., $\alpha\mathbf{P}$ and $(1-\alpha)\mathbf{v}$. Matrix $\mathbf{P}$ is partitioned in the Row scheme and Vector $\mathbf{x}$ and $\mathbf{v}$ are partitioned in the Broadcast scheme (from Table 3.4) because $\mathbf{x}$ and $\mathbf{v}$ are matrices of small sizes (two vectors). Table 4.4 lists all the statistics of the social network datasets used for the PageRank computation. All the graphs are sparse, and both MatFast(opt) and MatFast compute sparse matrix multiplications in compressed format. Figure 3.6a gives the average execution time for one iteration of PageRank on various social graph datasets. MatFast consistently performs the best. MatFast(opt) outperforms MatFast when rule-based heuristics are turned on and optimized matrix data partitioning schemes are adopted. For Dataset Twitter2010, the average execution time per iteration in MatFast(opt) is about 340s, while it needs more than 1000s for MatFast, 1800s for MLlib and 26000s for SystemML in Spark mode. MatFast(opt) caches the invariant matrices in the distributed memory without repetitive computations. Figure 3.6b shows that MatFast(opt) incurs lowest shuffle costs during each iteration. MatFast(opt) outperforms the MLlib and SystemML due to the following reasons: (1) it identifies and extracts the loop invari-



(a) Execution time

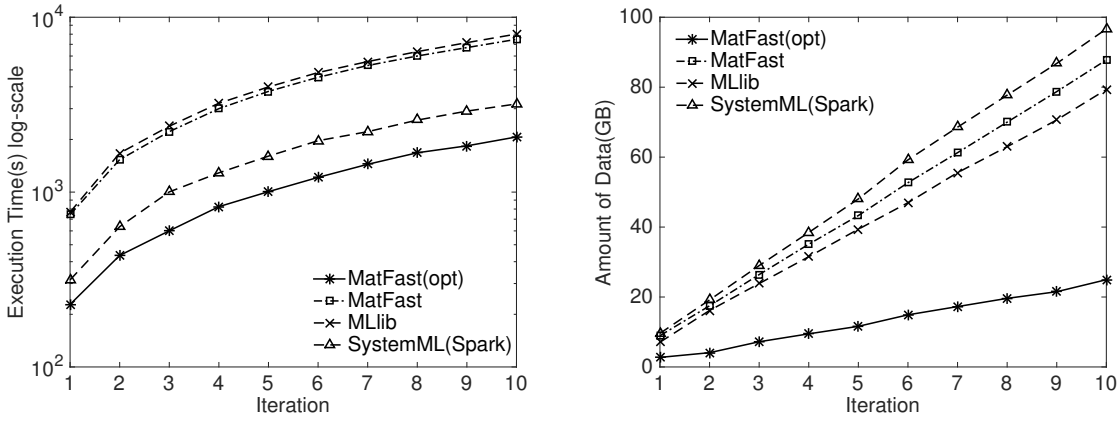
(b) Communication cost

Figure 3.6.: PageRank on different real-world datasets.

ant expressions, and caches the invariants without recomputing, and (2) it broadcasts the updated PageRank vector based on the cost-efficient partitioning scheme.

3.6.2 GNMF

The Netflix dataset consists of 100,480,507 ratings on 17,770 movies from 480,189 customers. In the experiment, the number of topics p is set to 200. The performance of GNMF is optimized by folding element-wise matrix operators and choosing a cost-efficient order for matrix chain multiplications. Figure 3.7a gives the accumulated execution time for the Netflix dataset over different systems. MatFast(opt) consistently performs the best, followed by SystemML and MLlib. The accumulated execution time of MatFast is very close to that of MLlib. When all the optimization strategies are turned off, MatFast computes a matrix chain multiplication in the sequential order that is exactly the same order adopted by MLlib. The slight performance improvement is due to MatFast using replication-based matrix multiplications for tiny matrices and cross-product based matrix multiplications for other input matrix types. Both MatFast(opt) and SystemML optimize the dense matrix



(a) Execution time

(b) Communication cost

Figure 3.7.: GNMF on the Netflix dataset.

chain multiplications. MatFast(opt)'s extra speedup is due to its optimized matrix data partitioning schemes.

Figure 3.7b gives the accumulated communication costs for the Netflix dataset over various systems. MatFast(opt) has minimal communication overhead due to its effective data partitioning schemes. SystemML does not capture data dependencies among different matrix operators. Thus, it incurs high communication overhead. MLlib fails to identify a good execution plan for matrix chain multiplications, and lacks efficient data partitioning schemes for matrix data. When the data partitioning scheme assignment is turned off for MatFast, the communication overhead is similar to that of SystemML. The reason is that MatFast adopts similar matrix multiplications strategies as SystemML.

3.6.3 BFGS

BFGS is widely used for solving unconstrained nonlinear optimization problems. For the Netflix problem, to obtain a good approximation of $\mathbf{V} \approx \mathbf{W} \times \mathbf{H}$, we can define an objective function as,

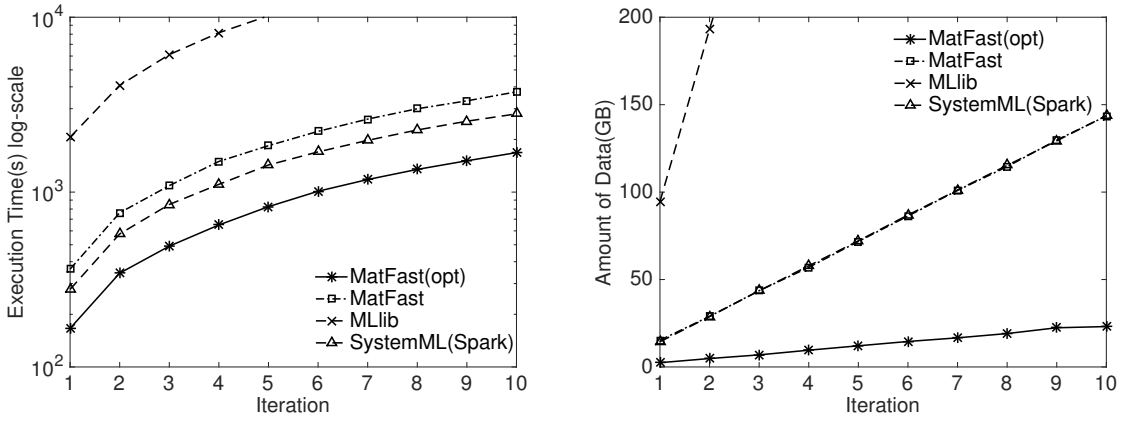
$$f(\mathbf{W}, \mathbf{H}) = ||\mathbf{V} - \mathbf{W} \times \mathbf{H}||_F^2 + ||\mathbf{W}||_F^2 + ||\mathbf{H}||_F^2,$$

where $||\mathbf{A}||_F^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}^2$. The gradient of $f(\mathbf{W}, \mathbf{H})$ is computed by the partial derivatives w.r.t. $\mathbf{W}$ and $\mathbf{H}$, i.e.,

$$\frac{\partial}{\partial \mathbf{W}} f(\mathbf{W}, \mathbf{H}) = 2\mathbf{W} - 2(\mathbf{V} - \mathbf{W} \times \mathbf{H}) \times \mathbf{H}^T,$$
$$\frac{\partial}{\partial \mathbf{H}} f(\mathbf{W}, \mathbf{H}) = 2\mathbf{H} - 2\mathbf{W}^T \times (\mathbf{V} - \mathbf{W} \times \mathbf{H}).$$

A complete gradient for all the variables is derived by concatenating the two vectorized partial derivatives. We implement the key update computation for BFGS on the Netflix dataset.

Figure 3.8a shows the accumulated execution time for BFGS update on the various systems. MatFast(opt) spends about 168s for each iteration using the low-rank matrix update heuristic. By turning off the optimization, MatFast spends about 372s for each iteration by storing the intermediate product matrices. SystemML performs slightly better than MatFast, and spends about 292s for each iteration. MLlib performs the worst and spends more than 2000s for a single iteration. The reason is that MLlib uses an inefficient strategy for matrix multiplications by duplicating copies of the input matrices. Figure 3.8b gives the communication overhead for each system. MatFast(opt) shuffles about 2.4GB data during each iteration to broadcast the updated gradient. Without the optimization for low-rank matrix update, MatFast shuffles about 14.4GB data to store intermediate results. SystemML and MatFast generate a similar amount of data shuffle. MLlib performs the worst due to its inefficient implementation of matrix multiplications.



(a) Execution time

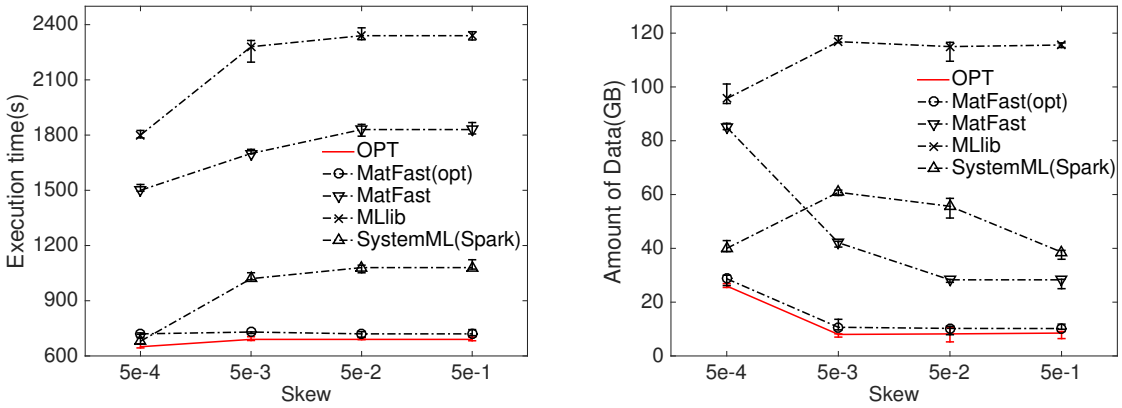
(b) Communication cost

Figure 3.8.: BFGS on the Netflix dataset.

3.6.4 Sparse matrix chain multiplication

We generate random matrices with the same dimensions to study the effectiveness of the dynamic cost-based optimization strategy. We generate matrix chains of length 4 ($\mathbf{A}_1 \times \mathbf{A}_2 \times \mathbf{A}_3 \times \mathbf{A}_4$). Each matrix is of size 30,000 $\times$ 30,000. The skew type of input Matrix $\mathbf{A}_1$ and $\mathbf{A}_3$ are set for column-major fashion, and those of Matrix $\mathbf{A}_2$ and $\mathbf{A}_4$ are set for row-major fashion.

We fix the sparsity at $\rho = 0.01$ and vary the skew of the generated matrix data. Figure 3.9 gives the execution time and communication cost with median, minimum, and maximum w.r.t. various skew values. OPT is the optimal plan by enumerating all possible execution plans. MLlib evaluates the sparse matrix chain multiplication in a sequential order. It incurs high compute time and communication overheads due to its inability to perform sparse matrix multiplications in compressed format. MatFast with the optimizations turned off conducts sparse matrix chain multiplications in a sequential order. It outperforms MLlib because all the sparse matrix operations are executed over the compressed format. SystemML exploits worst-case estimation to predict the sparsity of the intermediate results. When the skew is small, i.e., $s = 5e$-4,



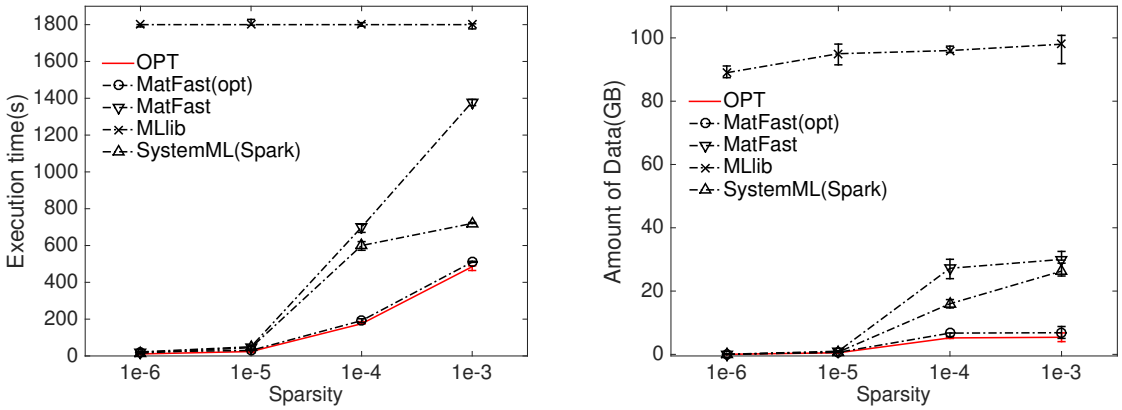
(a) Execution time

(b) Communication cost

Figure 3.9.: Costs of various skews for sparse matrix multiplication chain of length 4 with fixed sparsity $\rho = 0.01$.

the non-zero entries follow almost uniform distribution in the input matrices. The cost of the sequential execution plan is very close to the optimal. It takes MatFast(opt) extra overhead to collect the sampling statistics. Thus, MatFast(opt) performs worse than SystemML for extremely small skew values. However, dynamic cost estimation pays off as the skew becomes prominent. When the skew $s \geq 5e$-3, the optimal execution is no longer sequential. MatFast(opt) executes the multiplications in a nearly optimal order with negligible sampling costs.

Next, we fix the skew at $s = 0.5$ and vary the sparsity values. Figure 3.10 illustrates that when the sparsity value is very small, i.e., $\rho = 1e$-6, different plans have similar execution costs due to the extremely low sparsity in the matrices. MLlib performs the worst because it does not support sparse matrix multiplications in compressed format. As the sparsity increases, MatFast(opt) consistently outperforms SystemML with less execution time and communication cost. MatFast with the optimizations turned off performs slightly worse than SystemML when $\rho < 1e$-4. The sequential execution order adopted by MatFast incurs extensive computation costs when $\rho > 1e$-4 compared to SystemML.



(a) Execution time

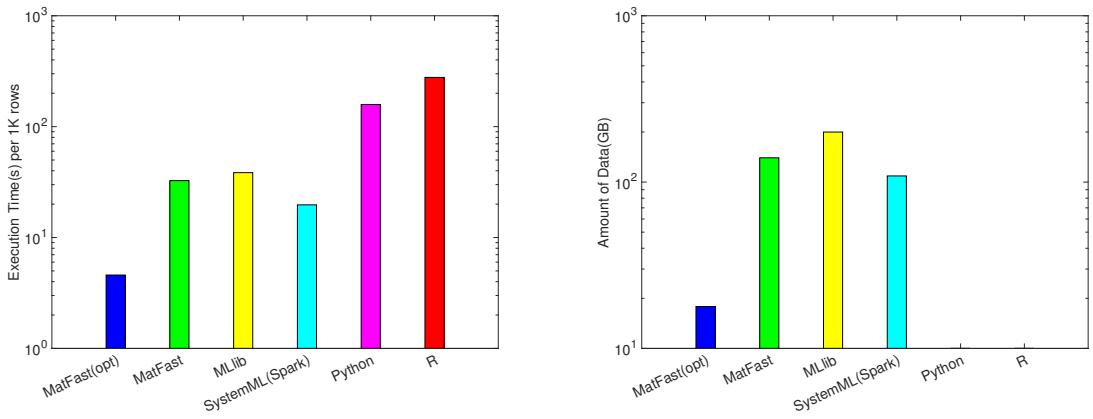
(b) Communication cost

Figure 3.10.: Costs of different sparsity values for matrix multiplication chains of length 4 with fixed skew $s = 0.5$.

3.6.5 Biological data analysis

For a comprehensive study of the proposed optimization techniques, we evaluate matrix computations in a complex biological data analysis (Expression Quantitative Trait Loci (eQTL)) using the kruX technique [20]. To perform eQTL analysis on a real-world genome dataset, we apply kruX on 462 samples. The size of the genotype matrix is 38,187,570 $\times$ 462, and the size of the expression matrix is 23,722 $\times$ 462. The output is a dense matrix (23,722 $\times$ 38,187,570) that takes about 7,250 GB of memory. This is far beyond our clusters' hardware. Thus, the whole genotype matrix is partitioned into chunks of size of $170,000 \times 462$, and the result takes about 32GB per chunk.

The kruX library provides serial implementations on popular platforms, e.g., Python and R. Figure 3.11 gives a performance comparison between the different platforms. We measure performance in terms of execution time per 1K rows of the result matrix. This metric is an important parameter for kruX to produce human checkable results. Both Python and R implementations run in a single node and do



(a) Execution time

(b) Communication cost

Figure 3.11.: kruX algorithm for eQTL over multiple platforms.

not leverage parallel compute resources. Python and R take 158s and 278s to produce 1K rows, respectively. In contrast, MatFast(opt) takes about 4.6s to compute 1K rows, while MatFast, MLlib and SystemML take 33s, 38.5s, and 20s, respectively, to compute 1K rows. The kruX algorithm involves various matrix operators, e.g., dense matrix multiplication with sparse matrix, matrix transpose, and element-wise operations. With an optimized execution plan and matrix data partitioning schemes, MatFast(opt) outperforms MLlib by 9 times and SystemML by 5 times for computing eQTL. MatFast(opt)'s communication cost is less than 10% of those for MLlib and SystemML.

3.6.6 Comparison with Non-MapReduce-based Systems

To complete the performance study, we also compare MatFast with non-MapReduce-based systems, i.e., ScaLAPACK and SciDB. ScaLAPACK [60] is a library for high-performance linear algebra routines on distributed-memory message-passing computers. It provides interfaces for distributed matrix computation. SciDB [59] is a scalable database management system designed for advanced analytics

on multi-dimensional array data model. SciDB provides SQL-like query interfaces for performing matrix operations, e.g., matrix multiplication and transpose.

Table 3.6.: Comparison with ScaLAPACK and SciDB

Operation	**ScaLAPACK**	**SciDB**	**MatFast(opt)**
MG-dense	103s	706s	118s
MG-sparse	86s	566s	19s

We use matrix-matrix multiplications for comparing the performance of the various systems. We select a dense partition $\mathbf{G}$ from the genotype matrix with dimension 10,000 $\times$ 462, and the whole mRNA sampling Matrix $\mathbf{M}$ with dimension 23,722 $\times$ 462. To generate a sparse Matrix $\mathbf{G}_s$, we fix the sparsity to 0.01 and randomly select each entry in $\mathbf{G}$ with probability of 0.01 to fill out the corresponding position in $\mathbf{G}_s$. For SciDB, the sparse matrix multiplication is computed with the `spgemm` operator. ScaLAPACK, SciDB, and MatFast all run on the Hathi cluster, where each node launches eight processes.

Row "MG-dense" in Table 3.6 gives the execution time for computing $\mathbf{M} \times \mathbf{G}^T$. ScaLAPACK and MatFast(opt)'s performance are comparable for dense matrix multiplications. However, ScaLAPACK does not provide any fault tolerance guarantees for big data computations. MatFast is built on top of Spark that naturally supports fault tolerance. SciDB is slower than the other two systems because it takes care of data placement on disk. SciDB redistributes the data on the compute workers to satisfy the requirement of ScaLAPACK. SciDB also incurs extra overhead for maintaining failure handling during execution. Row "MG-sparse" gives the execution time for computing $\mathbf{M} \times \mathbf{G}_s^T$. MatFast(opt) performs significantly better than ScaLAPACK and SciDB as it performs sparse matrix multiplication in compressed format. Both ScaLAPACK and SciDB are not well tuned for sparse matrix operations and they treat sparse matrices as dense ones.

3.7 Concluding Remarks

In this chapter, we present MatFast, an in-memory distributed platform that optimizes query pipelines of matrix operations. MatFast takes advantage of both dynamic cost-based analysis and rule-based heuristics to generate an optimized query execution plan. The dynamic cost-based analysis leverages a sampling-based technique to estimate the sparsity of matrix chain multiplications. The rule-based heuristics explore the special features of a matrix program and these features are organized in a memory-efficient way. Furthermore, communication-efficient data partitioning schemes are applied to input and intermediate matrices based on a cost function for matrix programs. MatFast has been implemented as a Spark library. The case studies and experiments on various matrix programs demonstrate that MatFast achieves an order of magnitude performance gain compared to state-of-the-art systems.

4 BIG-DATA QUERY PROCESSING FOR QUERIES THAT INVOLVE BOTH RELATIONAL AND MATRIX OPERATORS

4.1 Introduction

Data analytics, including machine learning (ML) and scientific research, often need to analyze large volumes of matrix data in various applications, e.g., self-driving cars [61], natural language processing [62], and social network analysis, recommender systems [63]. As ML models increase in complexity, and data volume accumulates drastically, traditional single-node solutions are incapable of processing this data efficiently. As a result, a number of distributed systems have been built to optimize data processing pipelines in a unified ecosystem of big matrix data for various kinds of applications.

In addition, queries to ML and scientific applications exhibit quite different characteristics from traditional business data processing applications. Complex analytics, e.g., linear regression (LR) for classification, and principle component analysis (PCA) for dimension reduction, are prevalent in these query workloads. These complex analytics replace the traditional SQL aggregations that are commonly used in traditional business intelligence applications. These analytical tasks rely heavily on large-scale matrix computations and are more CPU-intensive than the other traditional RDBMS computations.

The advent of MapReduce [33] and Apache Spark [51] has spurred a number of distributed matrix computation systems, e.g., Mahout [35], SystemML [36], DMac [41], MLlib [40], and MatFast [1]. In contrast to popular scientific platforms, e.g., R [37], the above systems provide better scalability and fault-tolerance in addition to efficiency in computations. However, these systems only focus on the efficient execution of *pure* matrix computations.

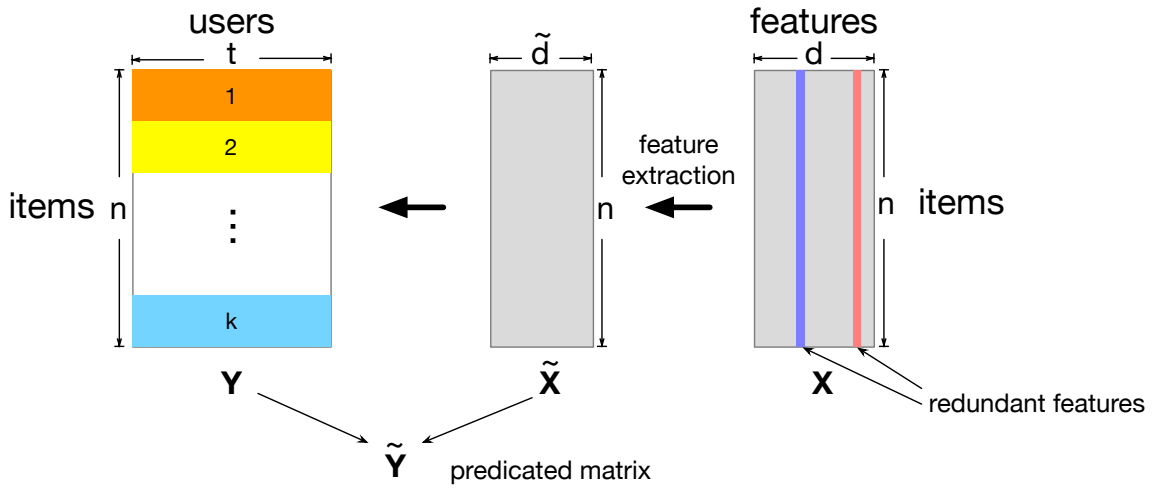


Figure 4.1.: Collaborative filtering with side information.

Data analytics on real-world applications need an entire data processing pipeline that usually consists of multiple stages and various data operations, e.g., relational selections and aggregations on matrix data. A typical pipeline of ML and scientific applications consists of several components, e.g., data pre-processing, core computation, and post-processing.

Example 1 (Collaborative Filtering) *Consider collaborative filtering with side information [64]. In Figure 4.1, the input is an n-by-t item-user recommendation matrix* $\mathbf{Y}$ *that contains the observed recommendation activities of t users over n items. An observed entry* Y_{ij} *indicates that the i-th item has been recommended by the j-th user. An unobserved entry* $Y_{ij} = 0$ *indicates an unknown relationship between the i-th item and the j-th user. As side information, the model assumes that there is an n-by-d item-feature Matrix* $\mathbf{X}$*, where each row represents a feature vector for the corresponding item. The goal is to identify the most promising potential items, say the top-k items, for each target user from her non-recommended items by leveraging both Matrix* $\mathbf{Y}$ *and* $\mathbf{X}$*. Before computing the optimal model parameters, one needs to construct Matrix* $\mathbf{X}$*, which could be obtained by crawling each item's webpage. Certain features may contain inconsistent or empty values. For successful feature extraction,*

all the empty columns should be removed from $\mathbf{X}$ *and any inconsistent data should be corrected by a data cleaning toolkit. Cross-validation [23] is a common technique to estimate the regularizers to prevent overfitting. The* k*-fold cross-validation divides the training dataset into* k *disjoint partitions, selects* $(k-1)$ *partitions as the training set, and keeps the* k*-th partition for testing. The generation of* k *disjoint partitions can be achieved by relational selections on the row dimension of Matrix* $\mathbf{Y}$*. The selected nonzero entries serve as the test set to evaluate the regularizers. For post-processing, a max aggregation is conducted on the predicated matrix* $\tilde{\mathbf{Y}}$*, which is computed from the collaborative filtering model based on Matrix* $\mathbf{Y}$ *and* $\tilde{\mathbf{X}}$*, to find the entries with the largest predicted values, and recommend them to potential users. This example suggests successful training and prediction of a complex ML model requires a seamless execution of high-performance relational operations and linear algebra operations.*

Although several systems provide efficient matrix computations, they do not offer relational operations over matrix data, or only provide premature relational support with limited optimizations. SciDB [59] is a full-fledged array database system that supports rich relational operations on array data. However, SciDB is not well-tuned for sparse matrix computations, and incurs extra overhead for maintaining failure handling during execution [41]. For other systems built on general dataflow platforms, e.g., SytemML, MLlib, MatFast, the relational operations could become a bottleneck in the entire matrix data processing pipeline if they are not treated properly. An effective matrix query optimizer should be able to process both relational and matrix operations efficiently, and discover the potential to execute the mixed types of operations interchangeably, e.g., pushing a relational operation under a matrix operation in the logical query evaluation plan with correctness guarantees.

In this chapter, we identify a series of equivalent transformation rules for rewriting logical query plans that involve both relational and matrix operations. For example, a query optimizer that is not sensitive to matrix operations evaluates $\Gamma_{\texttt{sum},r}(\mathbf{A}\times(\mathbf{B}+\mathbf{C}))$ by first computing the intermediate summation of $\mathbf{B}$ and $\mathbf{C}$, then conducting matrix-matrix multiplication on $\mathbf{A}$ and the intermediate sum, and finally performing the `sum`

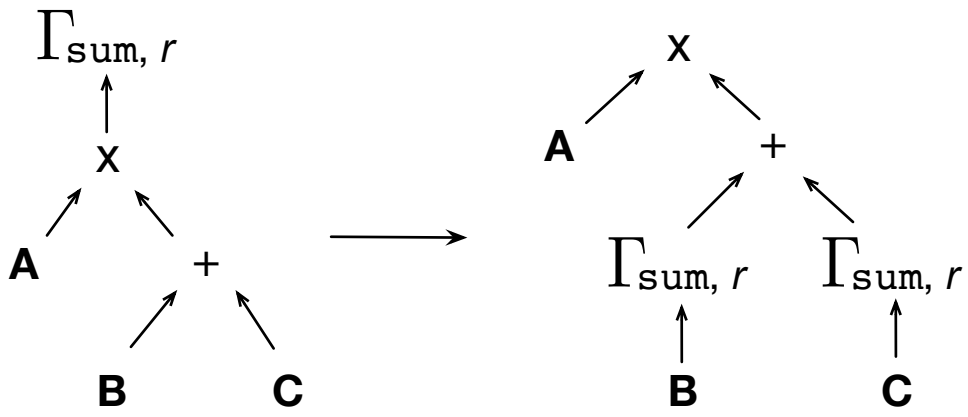


Figure 4.2.: Pushing aggregation under matrix multiplication.

aggregation along the row direction of the product matrix. An improved logical plan can be obtained by pushing the aggregation function below the matrix multiplication. Figure 4.2 gives two different logical plans. The second plan is more beneficial because the dimensions of the result are much smaller than the dimensions of the inputs after the aggregation. The smaller aggregated matrices further enhance the execution of the matrix-matrix multiplications. Thus, it is of great importance for a query optimizer to enumerate all the equivalent transformations, and pick the ones with the minimum computation overhead.

For a complex query pipeline consisting of both relational and matrix operations, optimizing computation overhead alone is not enough. Communication overhead incurred by expensive relational operations on big matrix data may dominate overall query executions. In a distributed setup, the communication overhead may become a bottleneck for complex relational operations, e.g., joins. Load-balanced matrix data partitioning schemes, e.g., hash-based schemes, may not be efficient for joins with data dependencies. Consider the collaborative filtering example. The predicted item-user recommendation matrix $\tilde{\mathbf{Y}}$ could be computed from two factor matrices $\mathbf{W}$ and $\mathbf{H}$ learned from the observed data $\mathbf{Y}$ and $\tilde{\mathbf{X}}$, e.g., $\tilde{\mathbf{Y}} = \mathbf{W} \times \mathbf{H}^T$. Suppose we have another user-item matrix $\mathbf{C}$ with the same users as $\tilde{\mathbf{Y}}$ on different items. Tensor

decomposition [31] allows to capture connections among the users and two different sets of items. Such a tensor is constructed by conducting a join operation on two matrices, i.e., $\tilde{\mathbf{Y}} \bowtie \mathbf{C}$, or $(\mathbf{W} \times \mathbf{H}^T) \bowtie \mathbf{C}$, on the column dimension of $\tilde{\mathbf{Y}}$ and the row dimension of $\mathbf{C}$. An efficient execution plan would partition Matrix $\tilde{\mathbf{Y}}$ in the column dimension and Matrix $\mathbf{C}$ in the row dimension to satisfy the requirement of the join predicate. Partitioning Matrix $\tilde{\mathbf{Y}}$ in the column dimension further requires evaluating the costs of different matrix data partitioning strategies on $\mathbf{W}$ and $\mathbf{H}$, and choosing the one with the minimum communication overhead. Therefore, optimizing the data partitioning of the input and intermediate matrices is also a critical step for efficiently evaluating relational operations over big matrix data.

In this chapter, we introduce MatRel, an in-memory distributed relational query processing system on big matrix data. MatRel has (1) a query optimizer to identify and leverage a series of transformation rules of interleaved relational and matrix operations to reduce the computation cost and the memory footprint, and (2) a matrix data partitioner to mitigate the communication overhead for the relational join operations on matrix data. The query optimizer utilizes rule-based query rewrite to generate an optimized logical plan. MatRel leverages a cost model to partition the input matrices for relational join operations to minimize communication overhead. MatRel is designed as an extension to Spark SQL that leverages existing relational query optimization techniques and dataflow operators.

The main contributions of this chapter are as follows:

- We develop MatRel, a system for efficient relational query processing over big matrix data in a distributed in-memory environment.
- We identify a series of equivalent transformation rules to rewrite a logical plan when both relational and matrix operations are present.
- We formally define the join operator over matrix data and propose corresponding optimization techniques to enhance runtime cost.

- We introduce a cost model to distribute matrix data among various workers for communication-efficient evaluation of relational join operations.
- We evaluate MatRel against state-of-the-art distributed matrix computation systems using real and synthetic datasets. Experimental results illustrate up to two orders of magnitude enhancement in performance.

The rest of this chapter proceeds as follows. Section 4.2 gives an overview of the supported matrix operators and MatRel's architecture. Section 4.3 presents the formal definitions of all the supported relational operations and their corresponding optimizations, e.g., relational selection, projection, aggregation, and join over matrix data. Section 4.4 discusses the local execution of various operators on the workers and system implementation. Section 4.5 presents the related work. Section 4.6 presents experimental results on various datasets and applications. Finally, Section 4.7 concludes the chapter.

4.2 An Overview of Distributed Relational Query Processing over Big Matrix Data

To facilitate relational query processing on matrix data, we realize a query optimizer for evaluating and optimizing a mixture of relational and matrix operators over distributed matrix data. To guarantee meaningful outputs from relational operators, the optimizer automatically infers the schema of the result given the input matrices and operators. Furthermore, we develop a cost model to decide on how the output matrix (tensor) data is partitioned, especially for the join operators. Each worker adopts a block-based matrix storage to conduct computations locally.

Figure 4.3 shows the major components of MatRel. We briefly introduce each component in the top-down order. The input to MatRel can be an ML algorithm or a graph analytical task that consists of various relational and matrix operators over the input matrices. The query can be expressed by the customized `DataFrame`, which is a wrapper of `Dataset[Row]`. A `DataFrame` provides basic relational and matrix operators to manipulate the input matrices, which is similar to the R pro-

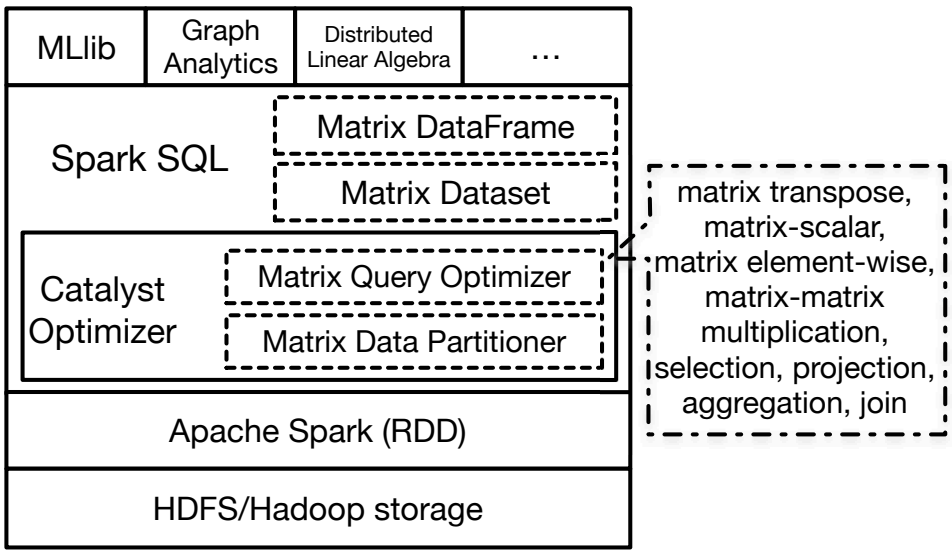


Figure 4.3.: Architecture of MatRel.

graming language [37]. A `DataFrame` organizes the operators as a logical evaluation plan, and the plan is fed to MatRel's optimizer for query optimization. MatRel's optimizer extends Spark SQL's Catalyst optimizer [65] by introducing new data types for matrix data. The optimizer is able to recognize matrix data and its schema. By leveraging rule-based query rewrite heuristics, the initial query plan is transformed to an optimized execution plan in a distributed environment. The optimizer invokes the matrix data partitioner to assign efficient partitioning schemes to the distributed matrix data based on a cost model. Internally, the matrix data is organized as Resilient Distributed Datasets (RDDs) [51] for distributed execution. For efficient and fault-tolerant storage of matrix data, MatRel leverages Hadoop Distributed File System (HDFS) for external storage.

4.3 Relational Operators on Matrix Data

In this section, we discuss the *conceptual* schema of a matrix in terms of relational algebra. Based on the well-defined schema, we introduce basic relational operators on

matrix data, i.e., selection, projection, aggregation, and join. We also discuss various optimization strategies for the relational operations over matrix data.

4.3.1 Relational Algebra on Matrix Data

Relational algebra [66] and the primitive operators have well-founded semantics for relations. All the relational operators assume the input data has a properly defined schema. The relational operations produce a closure on the relations, i.e., the output of a relational operator always generates a new relation with a certain schema, which is derived from the schema of the input. To ensure expressive relational operators over matrix data, we need to define a general schema, which captures the logical structure of data in a matrix.

Schema of a Matrix

A matrix **A** consists of two dimensional entries, A_{ij}'s. As an analogy, each entry in a matrix resembles a tuple in a table, and a matrix can be modeled by a relation naturally. Therefore, a general schema for a matrix can be defined as,

```
matrixA (RID, CID, val),
```

where `RID` is the row dimension, `CID` is the column dimension, and `val` is the value of the matrix entry. Attribute `val` may have its own schema as a tuple of multiple attributes. For the sake of simplicity, we only consider simple numerical numbers in the `val` attribute. In addition, MatRel also maintains the dimension of the matrix in the system catalog, i.e., the number of rows and the number of columns.

This schema specifies the logical structure of a matrix. However, it does *not* reflect the physical storage of the matrix data. For efficiency and storage consideration, a matrix is usually divided into "blocks" or "chunks", i.e., it is partitioned into smaller matrices, which can be moved around to specific compute nodes for high-performance local computations (refer to Section 3.4.1 for illustration).

4.3.2 Relational Selection on Matrix Data

Relational selection (σ) is a useful operator for picking the qualified entries from a matrix. We distinguish between selections on the dimensions, and the values of a matrix. Formally, a relational selection is a unary operator, written as

$$\sigma_{\theta(RID,CID,val)}(\mathbf{A}),$$

where θ is a propositional formula on the dimensions and entries of Matrix $\mathbf{A}$. An atom in θ is defined as $v\varphi c$ or $c\varphi v$, where $v \in \{RID, CID, val\}$ is one of the dimensions or the matrix entry, and c is a constant, $\varphi \in \{<, \leq, =, \neq, \geq, >\}$; and the logical expressions among atoms, i.e., $\wedge$ (and), $\vee$ (or), and $\neg$ (negation) expressions. A relational selection on a matrix produces another matrix, perhaps with different dimensions. The dimensions depend on the predicate, e.g., the predicate "$RID = i$" produces a matrix of a single row.

Selecting the Entries

We discuss different variants on the selection predicates, and corresponding optimization techniques. A selection outputs a matrix of the same dimension if the predicate only involves the entries. The entries that qualify the θ predicate are preserved, and the remaining are filled with 0's. An important property of the relational selection on matrix entries is that it does *not* change the dimension of the input. To optimize the execution, let us examine the components of the input. The input could be a matrix expression of a series of matrix operators, or one of multiple relational operators. There is little room in optimization for the former scenario. The input matrix has to be computed correctly before any selection is performed, as the selection predicate will be evaluated on the entries. In summary, we have the following optimization rule if the input matrix is composed of a series of selection operations,

$$\sigma_{\theta_1}(\sigma_{\theta_2}(\cdots(\sigma_{\theta_k}(\mathbf{A})))) = \sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_k}(\mathbf{A}), \tag{4.1}$$

where each θ_i is a predicate on the entries. For other relational operations, the selection has to wait until the evaluation is completed, e.g., join and aggregation operations.

Selecting the Dimensions

A selection predicate on dimensions is leveraged to pick slices of a matrix according to the given row/column dimension value. For example, cross-validation [23] is an ML technique to estimate how accurately a predicative model will perform in practice. Specifically, leave-*one*-out cross-validation can be realized by selecting a single row from a training matrix as the test set. We only consider the predicates involving equality conditions for the sake of simplicity, i.e., $\theta \in \{$"$RID = i$", "$CID = j$", "$RID = i \wedge CID = j$"$\}$.

The output of a selection on the dimensions is a matrix of different dimensions than the input. Suppose the input is an m-by-n Matrix $\mathbf{A}$, $\sigma_{RID=i}(\mathbf{A})$ generates a 1-by-n matrix, whose entries come from the i-th row of $\mathbf{A}$. Similarly, $\sigma_{CID=j}(\mathbf{A})$ outputs an m-by-1 matrix, and $\sigma_{RID=i \wedge CID=j}(\mathbf{A})$ outputs a 1-by-1 matrix, whose entries come from the j-th column, and the (i, j)-th entry of $\mathbf{A}$.

The selection predicate can be composed of conditions on both dimensions and entries. For instance, selecting all the entries whose values equal 10 from the 5-th row can be expressed as $\sigma_{val=10 \wedge RID=5}(\mathbf{A})$. Furthermore, in real-world ML applications, an empty column suggests no metrics have been observed for some feature. Removing empty columns will benefit the feature extraction procedure for better efficiency. Thus, we introduce two special predicates,

$$\sigma_{rows \neq 0}(\mathbf{A}) \quad \text{and} \quad \sigma_{cols \neq 0}(\mathbf{A}).$$

The former excludes the rows that are all 0's, and the latter excludes the columns that are all 0's. These are important extensions to the selection operator, since empty rows/columns usually do not contribute to matrix computations.

Optimizing Selection with Dimensions

A selection on dimensions exhibits potential opportunities for optimization. First, we discuss how to optimize a selection when the input is composed of matrix operations. Next, we explore optimizations on the selection when the input involves multiple relational operations.

MatRel supports multiple matrix operations, e.g., matrix transpose, matrix-scalar operation, matrix element-wise operation, and matrix-matrix multiplications. For matrix transpose, we have

$$\sigma_{RID=i}(\mathbf{A}^T) = (\sigma_{CID=i}(\mathbf{A}))^T, \tag{4.2}$$

$$\sigma_{CID=j}(\mathbf{A}^T) = (\sigma_{RID=j}(\mathbf{A}))^T. \tag{4.3}$$

These transformation rules reduce the computation costs significantly by avoiding the evaluation on matrix transposes. It takes $O(n^2)$ operations to compute the matrix transpose on an n-by-n matrix. However, it only takes $O(n)$ operations for a typical selection on a single dimension.

Similar rewrite rules apply to matrix-scalar and matrix element-wise operations as well. These rules can be viewed as pushing the selection below corresponding matrix operations:

$$\sigma_{RID=i}(\mathbf{A} + \beta) = \sigma_{RID=i}(\mathbf{A}) + \beta, \tag{4.4}$$

$$\sigma_{RID=i}(\mathbf{A} \star \mathbf{B}) = \sigma_{RID=i}(\mathbf{A}) \star \sigma_{RID=i}(\mathbf{B}), \tag{4.5}$$

where $\star \in \{+, *, /\}$. Rule 4.4 also applies to matrix-scalar multiplications. We illustrate these transformation rules on the selections with respect to the row dimension. The rules apply to selections on the column dimension as well. The number of required operations drops from $O(n^2)$ to $O(n)$ by circumventing the intermediate matrix.

For matrix-matrix multiplications, we have

$$\sigma_{RID=i}(\mathbf{A} \times \mathbf{B}) = \sigma_{RID=i}(\mathbf{A}) \times \mathbf{B}, \tag{4.6}$$

$$\sigma_{CID=j}(\mathbf{A} \times \mathbf{B}) = \mathbf{A} \times \sigma_{CID=j}(\mathbf{B}). \tag{4.7}$$

Proof (Rule 4.6) Let $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where $C_{ij} = \sum_k A_{ik} * B_{kj}$. $C_{i:}$ denotes the i-th row in Matrix $\mathbf{C}$. Then, by definition, $C_{i:} = \sum_k A_{ik} * B_{k:}$, where A_{ik} is the (i,k)-th entry from $\mathbf{A}$, and $B_{k:}$ is k-th row from $\mathbf{B}$. This can be viewed as the summation of each entry in the i-th row of $\mathbf{A}$ multiplying with corresponding rows of $\mathbf{B}$, i.e., $\mathbf{C}_{i:} = \mathbf{A}_{i:} \times \mathbf{B}$. ■

The correctness of Rule 4.7 can be verified in a similar manner. It takes $O(n^3)$ operations for a typical matrix-matrix multiplication, while a matrix-vector multiplication usually costs $O(n^2)$ operations. These transformation rules serve as basic optimizing blocks for producing strategies when a selection is performed on multiple dimensions. For example,

$$\sigma_{RID=i \wedge CID=j}(\mathbf{A} \times \mathbf{B}) = \sigma_{RID=i}(\mathbf{A}) \times \sigma_{CID=j}(\mathbf{B}), \tag{4.8}$$

which reduces the complexity from $O(n^3)$ to $O(n)$ for the selection on an entry from a matrix-matrix multiplication to vector inner-product.

4.3.3 Projection on Matrix Data

Relational projection (Π) is used to select a subset of the attributes from the matrix. Formally, a projection on a matrix $\mathbf{A}$ can be written as

$$\Pi_{\tau(RID,CID,val)}(\mathbf{A}),$$

where τ is a subset of all the attributes $\{RID, CID, val\}$. When `val` is a tuple, the predicate τ can be used to project only the interesting attributes from `val`'s. The produced matrix has the same dimension as $\mathbf{A}$, and each entry contains the projected attributes from `val`. Some applications may only be interested in the entries of a matrix, without caring about the dimensions, e.g., estimating the distribution of matrix entries. A projection on the dimensions is less common, since the dimensions could be obtained from the matrix metadata directly.

The input to a projection can have different formats: a matrix expression of a series of matrix operations, or a matrix produced by relational operations. When a projection involves the entry, the input has to be evaluated before the projection. However, it is very efficient to evaluate a projection if it only involves dimensions of the input. No actual computation is required as the dimensionality can be inferred from the matrix operations or the relational operations. The only exception is the

selecting the nonzero rows/columns. The nonzero selection has to be evaluated as it may change the dimension of the matrix.

4.3.4 Aggregation on Matrix Data

Traditionally, an aggregation is defined on the columns of a relation with various aggregate functions, e.g., `sum` and `count`. The aggregation operator is also beneficial on matrix data, e.g., computing the average ratings of all the users in the user-movie rating database. The (i, j)-th entry in the user-movie rating matrix denotes the rating from $user_i$ on $movie_j$. This requires an average computation on the row dimension. Unlike relational data, aggregations on a matrix may occur in different dimensions. Formally, an aggregation is a unary operator, defined as

$$\Gamma_{\rho,dim}(\mathbf{A}),$$

where ρ is the name of the aggregate function, and *dim* indicates the chosen dimension, i.e., *dim* could be *row* (r), *column* (c), *diagonal* (d), or *all* (a). Specially, MatRel supports various aggregate functions, $\rho \in \{\texttt{sum}, \texttt{nnz}, \texttt{avg}, \texttt{max}, \texttt{min}\}$. We introduce each aggregate function and corresponding optimizations in this section.

`Sum()` on Matrix Data

For `sum` aggregation, MatRel supports four different variants, $\Gamma_{\texttt{sum},r/c/d/a}(\mathbf{A})$. We first introduce their definitions, then discuss optimization strategies. Given an m-by-n Matrix $\mathbf{A}$, $\Gamma_{\texttt{sum},r}(\mathbf{A})$ produces a m-by-1 matrix, or a column vector, whose i-th entry computes the summation along the i-th row of $\mathbf{A}$. Similarly, $\Gamma_{\texttt{sum},c}(\mathbf{A})$ generates a 1-by-n matrix, or a row vector, which computes the summations of entries along the column direction. $\Gamma_{\texttt{sum},d}(\mathbf{A})$ is defined only if $m = n$ for square matrices, which is also called the *trace* of a square matrix. The trace is simply a 1-by-1 matrix, or a scalar value. Finally, $\Gamma_{\texttt{sum},a}(\mathbf{A})$ computes the summation of all the entries.

We first present transformation rules for optimizing `sum` aggregation when the input consists of various matrix operations. If the input is a matrix transpose, we have

$$\Gamma_{\texttt{sum},r/c}(\mathbf{A}^T) = (\Gamma_{\texttt{sum},c/r}(\mathbf{A}))^T, \tag{4.9}$$

$$\Gamma_{\texttt{sum},d/a}(\mathbf{A}^T) = \Gamma_{\texttt{sum},d/a}(\mathbf{A}), \tag{4.10}$$

By pushing a `sum` aggregation before a matrix transpose, MatRel computes the transpose on a matrix of smaller dimensions. For trace and all entry summation, the evaluation of the transpose can be avoided.

For matrix-scalar operations, we can derive the following transformation rules,

$$\Gamma_{\texttt{sum},r/c/a}(\mathbf{A} + \beta) = \Gamma_{\texttt{sum},r/c/a}(\mathbf{A}) + \beta n/\beta m/\beta mn, \tag{4.11}$$

$$\Gamma_{\texttt{sum},d}(\mathbf{A} + \beta) = \Gamma_{\texttt{sum},d}(\mathbf{A}) + \beta n, \tag{4.12}$$

$$\Gamma_{\texttt{sum},\cdot}(\mathbf{A} * \beta) = \Gamma_{\texttt{sum},\cdot}(\mathbf{A}) * \beta, \tag{4.13}$$

where Rule 4.12 only holds for square matrices, i.e., $m = n$. For matrix-scalar multiplications, similar rules apply when the aggregation is executed along the column direction, diagonal direction, and the entire matrix.

For matrix element-wise operations, only the matrix element-wise addition has a compatible transformation rule for `sum` aggregations.

$$\Gamma_{\texttt{sum},\cdot}(\mathbf{A} + \mathbf{B}) = \Gamma_{\texttt{sum},\cdot}(\mathbf{A}) + \Gamma_{\texttt{sum},\cdot}(\mathbf{B}). \tag{4.14}$$

Though Rule 4.14 does not change the computation overhead, it circumvents storing the intermediate sum matrix $\mathbf{A} + \mathbf{B}$, alleviating memory footprint.

For matrix-matrix multiplications, we derive efficient transformation rules to compute the `sum` aggregations,

$$\Gamma_{\texttt{sum},r}(\mathbf{A} \times \mathbf{B}) = \mathbf{A} \times \Gamma_{\texttt{sum},r}(\mathbf{B}), \tag{4.15}$$

$$\Gamma_{\texttt{sum},c}(\mathbf{A} \times \mathbf{B}) = \Gamma_{\texttt{sum},c}(\mathbf{A}) \times \mathbf{B}, \tag{4.16}$$

$$\Gamma_{\texttt{sum},a}(\mathbf{A} \times \mathbf{B}) = \Gamma_{\texttt{sum},c}(\mathbf{A}) \times \Gamma_{\texttt{sum},r}(\mathbf{B}), \tag{4.17}$$

$$\Gamma_{\texttt{sum},d}(\mathbf{A} \times \mathbf{B}) = \Gamma_{\texttt{sum},a}(\mathbf{A}^T * \mathbf{B}) = \Gamma_{\texttt{sum},a}(\mathbf{A} * \mathbf{B}^T). \tag{4.18}$$

Proof (Rule 4.15) Let $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where $C_{ij} = \sum_k A_{ik} * B_{kj}$. Let the column Vector $\mathbf{y} = \Gamma_{\mathtt{sum},r}(\mathbf{C})$. Therefore,

$$y_i = \sum_j C_{ij} = \sum_j \sum_k A_{ik} * B_{kj} = \sum_k A_{ik} * b_k,$$

where Vector $\mathbf{b} = \Gamma_{\mathtt{sum},r}(\mathbf{B})$. It is clear that the derivation in the matrix format, $\mathbf{y} = \mathbf{A} \times \mathbf{b}$, which is *exactly* Rule 4.15. ■

Similar derivations also apply to Rule 4.16 and 4.17. For Rule 4.18, both input matrices are required to be square.

Proof (Rule 4.18) Let us write Matrix $\mathbf{A}$ in terms of rows, and Matrix $\mathbf{B}$ in terms of columns,

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_n^T \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \cdots & \mathbf{b}_n \end{bmatrix}.$$

Therefore, the matrix-matrix multiplication can be represented by Vector $\mathbf{a}_i$'s and $\mathbf{b}_i$'s.

$$\mathbf{A} \times \mathbf{B} = \begin{bmatrix} \mathbf{a}_1^T\mathbf{b}_1 & \ldots & \mathbf{a}_1^T\mathbf{b}_n \\ \vdots & \ddots & \vdots \\ \mathbf{a}_n^T\mathbf{b}_1 & \ldots & \mathbf{a}_n^T\mathbf{b}_n \end{bmatrix}$$

The trace of $\mathbf{A} \times \mathbf{B}$ can be computed as $\Gamma_{\mathtt{sum},d}(\mathbf{A} \times \mathbf{B}) = \sum_{i=1}^n \mathbf{a}_i^T \times \mathbf{b}_i$. The transpose of $\mathbf{A}$ is represented as,

$$\mathbf{A}^T = \begin{bmatrix} \mathbf{a}_1 & \cdots & \mathbf{a}_n \end{bmatrix},$$

and the matrix element-wise multiplication is computed as,

$$\mathbf{A}^T * \mathbf{B} = \begin{bmatrix} \mathbf{a}_1 * \mathbf{b}_1 & \cdots & \mathbf{a}_n * \mathbf{b}_n \end{bmatrix}.$$

When conducting the all entry `sum` aggregation, it is essentially to compute the inner-product of corresponding vector pairs, i.e., $\Gamma_{\mathtt{sum},a}(\mathbf{A}^T * \mathbf{B}) = \sum_{i=1}^n \mathbf{a}_i^T \times \mathbf{b}_i$. The second half of Rule 4.18 can be verified in a similar manner. ■

The computation complexity is usually $O(n^3)$ for n-by-n square matrix-matrix multiplications. By leveraging matrix element-wise multiplication and matrix transpose, the complexity is reduced to $O(n^2)$.

Unfortunately, equivalent transformation rules do not apply when the input involves a relational selection or projection. For general selection condition θ and projection predicate τ,

$$\Gamma_{\texttt{sum},\cdot}(\sigma_\theta(\mathbf{A})) \neq \sigma_\theta(\Gamma_{\texttt{sum},\cdot}(\mathbf{A})), \tag{4.19}$$

$$\Gamma_{\texttt{sum},\cdot}(\Pi_\tau(\mathbf{A})) \neq \Pi_\tau(\Gamma_{\texttt{sum},\cdot}(\mathbf{A})). \tag{4.20}$$

However, there exist valid transformation rules if the `sum` dimension happens to match with the predicate of the selection, e.g., $\Gamma_{\texttt{sum},r}(\sigma_{RID=i}(\mathbf{A})) = \sigma_{RID=i}(\Gamma_{\texttt{sum},r}(\mathbf{A}))$. Swapping the execution order of `sum` aggregation and projection leads to the same result. Pushing a selective relational operation below a matrix operation is beneficial, as it could reduce the dimension of the matrix.

`Count()` on Matrix Data

For relational data, the `count` aggregate function computes the number of tuples that satisfy a certain criteria. For matrix data, the `count` function is usually interpreted as computing the number of nonzeros (`nnz`). This function is quite useful for optimizing matrix computations, e.g., sparsity estimation for sparse matrix chain multiplications [1]. Similar to `sum` aggregation, MatRel supports four `count` variants, $\Gamma_{\texttt{nnz},r/c/d/a}(\mathbf{A})$. The output of each function is defined as its counterpart of the `sum` function correspondingly.

We present equivalent transformation rules for optimizing `count` aggregations when the input is composed of various matrix operations. For matrix transpose, we have

$$\Gamma_{\texttt{nnz},r/c}(\mathbf{A}^T) = (\Gamma_{\texttt{nnz},c/r}(\mathbf{A}))^T, \tag{4.21}$$

$$\Gamma_{\texttt{nnz},d/a}(\mathbf{A}^T) = \Gamma_{\texttt{nnz},d/a}(\mathbf{A}). \tag{4.22}$$

These rules push a `count` aggregation below a matrix transpose. The postponed matrix transpose is executed on a smaller matrix, e.g., a vector, avoiding $O(n^2)$ computations for the matrix transpose.

For matrix-scalar operations, let $\mathbf{A}$ be an m-by-n matrix, and scalar $\beta \neq 0$. We denote $\mathbf{e}(n)$ to be an all-one column vector of length n. The following rules hold for an input that involves matrix-scalar operations,

$$\Gamma_{\mathtt{nnz},r}(\mathbf{A} + \beta) = \mathbf{e}(m) * n, \tag{4.23}$$

$$\Gamma_{\mathtt{nnz},c}(\mathbf{A} + \beta) = \mathbf{e}(n)^T * m, \tag{4.24}$$

$$\Gamma_{\mathtt{nnz},d}(\mathbf{A} + \beta) = n, \tag{4.25}$$

$$\Gamma_{\mathtt{nnz},a}(\mathbf{A} + \beta) = mn, \tag{4.26}$$

$$\Gamma_{\mathtt{nnz},\cdot}(\mathbf{A} * \beta) = \Gamma_{\mathtt{nnz},\cdot}(\mathbf{A}). \tag{4.27}$$

By assuming the constant $\beta \neq 0$, the number of zeros is not altered for matrix-scalar multiplications. For the matrix-scalar addition, the output does not depend on the entries in $\mathbf{A}$, which only requires the dimensions to conduct the computation.

For matrix element-wise operations, only the division operator works friendly with `count` aggregation.

$$\Gamma_{\mathtt{nnz},\cdot}(\mathbf{A}/\mathbf{B}) = \Gamma_{\mathtt{nnz},\cdot}(\mathbf{A}) \tag{4.28}$$

Rule 4.28 indicates the matrix element-wise division preserves the number of nonzeros of the numerator matrix. Unfortunately, such convenient transformation rules do not apply to other matrix element-wise operators and matrix-matrix multiplications.

For an input consisting of relational selections and projections, Rule 4.19 and 4.20 can be extended too. It is impossible to swap the execution order of a `count` aggregation and a selection for a correct output. It is only valid to swap the execution order of a `count` aggregation and a selection when both work on the same dimension, e.g., $\Gamma_{\mathtt{nnz},c}(\sigma_{CID=j}(\mathbf{A})) = \sigma_{CID=j}(\Gamma_{\mathtt{nnz},c}(\mathbf{A}))$.

`Avg()` on Matrix Data

The `avg` aggregate function computes the average statistic on a matrix in a certain

dimension. MatRel supports four different variants, $\Gamma_{\texttt{avg},r/c/d/a}(\mathbf{A})$. The output of each `avg` aggregate function is defined as its counterpart of the `sum` aggregation correspondingly. The `avg` aggregation can be viewed as a compound operator from a `sum` and a `count` aggregations. Formally, any `avg` aggregation can be computed as

$$\Gamma_{\texttt{avg},\cdot}(\mathbf{A}) = \Gamma_{\texttt{sum},\cdot}(\mathbf{A})/\Gamma_{\texttt{nnz},\cdot}(\mathbf{A}).$$

Therefore, optimizing an `avg` aggregation can leverage all the transformation rules we have explored for `sum` and `count` aggregations. MatRel's optimizer automatically invokes query planning for corresponding `sum` and `count` aggregations when generating an execution plan for an `avg` aggregation.

`Max()/Min()` on Matrix Data

The `max` (or `min`) aggregation is used to obtain the extreme values from a matrix, which serves as an important building block for anomaly detection [67] in data mining applications. Naturally, MatRel supports four `max` (or `min`) variants, $\Gamma_{\texttt{max/min},r/c/d/a}(\mathbf{A})$. The output of each aggregation is defined as its counterpart of the `sum` aggregation correspondingly. The transformation rules for optimizing `max` (or `min`) aggregation can be derived as follows,

$$\Gamma_{\texttt{max/min},r/c}(\mathbf{A}^T) = (\Gamma_{\texttt{max/min},c/r}(\mathbf{A}))^T, \tag{4.29}$$

$$\Gamma_{\texttt{max/min},d/a}(\mathbf{A}^T) = \Gamma_{\texttt{max/min},d/a}(\mathbf{A}), \tag{4.30}$$

$$\Gamma_{\texttt{max/min},\cdot}(\mathbf{A} + \beta) = \Gamma_{\texttt{max/min},\cdot}(\mathbf{A}) + \beta, \tag{4.31}$$

$$\Gamma_{\texttt{max/min},\cdot}(\mathbf{A} * \beta) = \Gamma_{\texttt{max/min},\cdot}(\mathbf{A}) * \beta \quad (\beta > 0), \tag{4.32}$$

$$\Gamma_{\texttt{max/min},\cdot}(\mathbf{A} * \beta) = \Gamma_{\texttt{min/max},\cdot}(\mathbf{A}) * \beta \quad (\beta < 0). \tag{4.33}$$

We assume the constant scalar $\beta \neq 0$. When $\beta < 0$, the optimizer computes the `min`/`max` aggregation instead for avoiding the intermediate matrix $\mathbf{A} * \beta$. For matrix element-wise operations and matrix-matrix multiplications, *no* general rules apply. For example, $\Gamma_{\texttt{min},r}([1,1,2] + [3,0.5,0]) = \Gamma_{\texttt{min},r}([4,1.5,2]) = 1.5$, which could only be evaluated after computing the sum of the two matrices. For an input consisting of relational selections and projections, it is only valid to swap the execution order of a `max`/`min` aggregation and selection when both work on the same dimensions.

4.3.5 Relational Join on Matrix Data

The relational join ($\bowtie$) is a useful operator for picking corresponding entries that satisfy the join predicates from two separate matrices. For example, raster data overlay analysis [68] requires a join on two matrices with matching row/column index, where a raster map can be interpreted as a matrix. Formally, a relational join is a binary operator,

$$\mathbf{A} \bowtie_{\gamma,f} \mathbf{B},$$

where γ is the join predicate, f is the user-defined merge function that takes two matching entries and outputs a merging result, e.g., $z = f(x, y)$. We introduce various formats of γ in different semantics in the subsequent sections. In general, the result of a relational join on two matrices is a *tensor*, or a multi-dimensional array, whose dimensions are determined by the predicate γ. We first discuss how to infer the schema of a join output based on the join predicates. Next, we introduce all the valid formats of γ. Finally, we investigate the optimization strategies for join executions.

Schema of Join Result

Just like every matrix has a schema, MatRel automatically produces a schema for a join result. The schema of a join result consists of two parts: the index and the value. In this work, we only consider equality predicates as the join condition. Given two input matrices $A(RID_A, CID_A, val_A)$ and $B(RID_B, CID_B, val_B)$, the cardinality of the dimension of $\mathbf{A} \bowtie_{\gamma,f} \mathbf{B}$ is

$$d = 4 - \delta_{dim},$$

where δ_{dim} is the number of equality predicates on the join dimensions. For example, if the join predicates γ = "$RID_A = RID_B$", the join output boasts the schema (D_1, D_2, D_3, val), where D_1 takes matching dimension from RID_A, D_2 takes relevant dimension from CID_A, D_3 takes relevant dimension from CID_B, and val is the evaluation of the merge function on the matching entries from $\mathbf{A}$ and $\mathbf{B}$. The join output may degrade to a normal matrix when γ contains two or more predicates on the join dimensions.

Cross-product on Two Matrices

The cross-product operator is a special join operator between two matrices, where the join predicate γ is empty, i.e., $\mathbf{A} \otimes \mathbf{B} = \mathbf{A} \bowtie_f \mathbf{B}$. If both input matrices are viewed as relations, this operator is essentially the Cartesian product. The cross-product is widely used in tensor decomposition and applications, e.g., Kronecker product [31]. MatRel's optimizer infers the output schema from the join predicate, which is empty for cross-product. Therefore, a cross-product produces a 4th-order tensor from two input matrices. For example, $C(D_1, D_2, D_3, D_4, val)$ is the schema of $\boldsymbol{\mathcal{C}} = \mathbf{A} \otimes \mathbf{B}$. The first two dimensions (D_1, D_2) inherit from $\mathbf{A}$, and the last two dimensions (D_3, D_4) inherit from $\mathbf{B}$.

The execution of a cross-product is conducted in a fully parallel manner. MatRel stores a matrix in blocks of equal size, where the blocks are distributed among all the workers in a cluster. First, each block of the smaller matrix is duplicated p times, where p is the number of partitions from the larger matrix. Next, each duplicated block is sent to a corresponding partition of the larger matrix. Finally, each worker performs join operation locally without further communication. The 4th-order tensor is stored as a series of block matrices in a distributed manner. The detailed physical storage is described in Section 4.4.

Join on Two Dimensions

The dimension-based predicates for join on two dimensions can take two different formats:

$$\mathbf{A} \bowtie_{RID_A=RID_B \wedge CID_A=CID_B, f} \mathbf{B},$$

or

$$\mathbf{A} \bowtie_{RID_A=CID_B \wedge CID_A=RID_B, f} \mathbf{B}.$$

These two predicates are the only valid formats, as a propositional formula takes dimensions from both inputs. The former predicate can be viewed as an overlay on the two input matrices directly (*direct overlay*), while the latter can be viewed as an overlay on Matrix $\mathbf{A}$ and the transpose of Matrix $\mathbf{B}$ (*transpose overlay*). Figure 4.4 illustrates an example of direct overlay. Conceptually, two input matrices can be

regarded as two relations with the schema introduced in Section 4.3.2. The output of a join on two dimensions is a normal matrix, since two common dimensions are shared with the inputs.

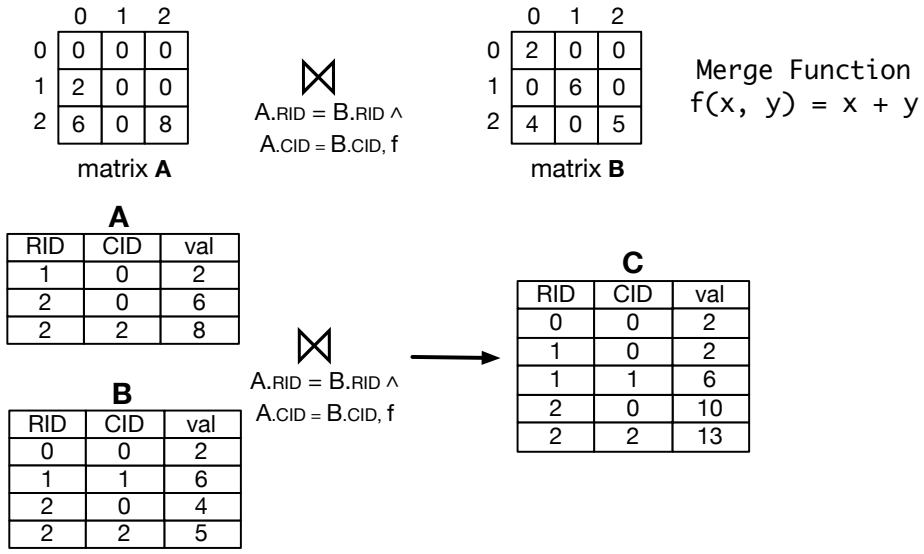


Figure 4.4.: Direct overlay of two sparse matrices.

To efficiently evaluate joins on two dimensions, MatRel adopts the hash join strategy to partition the matrix blocks from the inputs. By hashing the matched matrix blocks to the same worker, a worker conducts local join execution without further communications. For direct overlay, MatRel's planner partitions the two input matrices using the same partitioner. For transpose overlay, the planner makes sure the partitioning scheme on one input matches the partitioning scheme on the transpose of the other.

Join on a Single Dimension

The dimension-to-dimension (D2D) predicates can take four different formats:

$$\mathbf{A} \bowtie_{ID_A=ID_B,f} \mathbf{B},$$

where $ID_{A/B} \in \{RID_{A/B}, CID_{A/B}\}$. Therefore, the join output is a 3rd-order tensor, and takes the schema (D_1, D_2, D_3, val). Each entry from a matched row/column from $\mathbf{A}$ is joined with the entries from the corresponding row/column from $\mathbf{B}$.

MatRel leverages a hash-based data partitioner to map matched row or column matrix blocks to the same worker. Suppose both input matrices are partitioned into ℓ-by-ℓ square blocks. Let us consider the D2D predicate to be "$RID_A = RID_B$". Two matrix blocks $\mathbf{A}_{mp}$ and $\mathbf{B}_{mq}$ are mapped to the same worker. For the t-th row from $\mathbf{A}_{mp}$, each entry is joined with the same row from $\mathbf{B}_{mq}$. Thus, a 1-by-ℓ vector is produced by joining a single entry in the t-th row from $\mathbf{A}_{mp}$ and an entire row from $\mathbf{B}_{mq}$. After the join operation, the t-th row from $\mathbf{A}_{mp}$ leads to an ℓ-by-ℓ matrix block. Potentially, there are totally ℓ such matrix blocks for the join result from two input matrix blocks.

Join on Entries

The entry-based (value-to-value, or V2V) join operation takes the following format:

$$\mathbf{A} \bowtie_{val_A = val_B, f} \mathbf{B}.$$

The join output is a 4th-order tensor, where the first two dimensions come from the dimensions of $\mathbf{A}$, and the other two from the dimensions of $\mathbf{B}$.

The execution of a V2V join is conducted in a fully parallel manner, similar to the execution of cross-product. MatRel broadcasts each block of the smaller input to each partition of the larger one. Each worker in the cluster adopts a nested-loop join strategy to compute the join locally by taking each pair of matrix blocks. The four dimensions are copies of the dimension values from matched entries.

Join on a Single Dimension and an Entry

The join on a single dimension and an entry operation (D2V, or V2D) takes the following format:

$$\mathbf{A} \bowtie_{ID_A = val_B, f} \mathbf{B}, \text{ or } \mathbf{A} \bowtie_{val_A = ID_B, f} \mathbf{B}$$

where $ID_{A/B} \in \{RID_{A/B}, CID_{A/B}\}$. The output is a 4th-order tensor, where the 4 dimensions derive from the entry locations of the matched rows/columns, and the matched entries from the other input. To evaluate this type of join, the matrix blocks containing the matched entries are mapped to the other matrix, where the

row/column dimension matches. Each worker conducts local join computation based on the matched dimensions and entries.

Join Optimization

Relational joins over matrix data are more complex than other relational operations. They potentially have higher computation and communication overhead. We have identified several heuristics to mitigate the heavy memory footprint and computation burden. We leverage the following features: (1) sparsity-preserving merge functions, and (2) Bloom-join for join on entries. Furthermore, we also develop a cost model to capture the communication overhead among different joins. By utilizing the heuristics and the cost model, MatRel generates a computation- and communication-efficient execution plan for a join operation.

Identifying Sparsity-preserving Merge Functions.

Given a general join operation, $\mathbf{A} \bowtie_{\gamma,f} \mathbf{B}$, there are two important parameters, γ and f. The predicate γ is utilized to locate the join entries, and f is evaluated on the two entries for an output entry. In real-world applications, large matrices are usually sparse and structured[1]. For example, Figure 4.4 illustrates a direct overlay on two sparse matrices, where the merge function is $f(x, y) = x + y$. We say $f(x, y)$ is a sparsity-preserving function if $f(0, \cdot) = 0$ or $f(\cdot, 0) = 0$. Thus, the summation merge function is *not* sparsity-preserving since $C_{00} = A_{00} + B_{00} = 0 + 2 = 2$, which does not preserve sparsity from the left- or right-hand side. On the other hand, $f_1(x, y) = xy$, is sparsity-preserving on both sides.

The benefit of utilizing sparsity-preserving function is obvious, as it avoids evaluating the function completely if one of the inputs contains all zeros. Traditionally, a straw man execution plan takes two input matrix blocks. It at first converts the sparse matrix formats to the dense counterpart by explicitly filling in 0's, then evaluates the merge function on the matched entries. Note that a sparse matrix format only records nonzeros and their locations. Compressed sparse representations reduce memory consumption and computation overhead for matrix manipulations. However,

[1] `http://www.cise.ufl.edu/research/sparse`

the memory consumption grows significantly when a sparse matrix is stored in the dense format.

Thus, it is critical to identify the sparsity-preserving merge functions, and avoid converting a sparse matrix format to dense format whenever possible. We consider a special family of merge functions: linear function and their linear combinations, i.e., $f(x, y) = g(x)y + h(x)$, or $f(x, y) = g(y)x + h(y)$, where both $g(\cdot)$ and $h(\cdot)$ are linear functions. For any merge function in this family, MatRel adopts a sampling approach to identify the sparsity-preserving property. Given a merge function $f(x, y)$, we compute $t_1 = f(0, s_1)$, and $t_2 = f(0, s_2)$, where s_1 and s_2 are random numbers. If both t_1 and t_2 are 0, $f(x, y)$ is sparsity-preserving because of the linearity of function components. A similar sparsity-preserving test could be conducted on the y value.

Once a sparsity-preserving function is identified, MatRel's planner orchestrates the execution of join operations by leveraging the sparse matrix computations. For example, if the merge function $f(x, y)$ boasts the sparsity-preserving property on the x component, MatRel only retrieves the nonzero entries from the left-hand side, and joins with entries from the right-hand side. All the computations are conducted in the sparse matrix format without converting to dense matrix format.

Bloom-join on Entries.

For a join predicate that involves a dimension on the matrix, it is efficient for MatRel to locate the corresponding row matrix blocks or column matrix blocks. The irrelevant matrix blocks are not accessed during evaluation. However, the join operation becomes expensive when the join predicate contains a comparison on the entries. A straw man plan compares each pair of entries exhaustively, where a lot of computation is wasted on the mismatched entries.

MatRel adopts a Bloom-join strategy when a join predicate contains matrix entries. Each worker computes a Bloom filter on the entries. Due to the existence of sparse matrices, a worker needs to determine whether to store 0's in the Bloom filter. Thanks to the heuristic of identifying zero-preserving merge functions, 0 values are not inserted into the Bloom filter if $f(x, y)$ is sparsity-preserving. After each matrix

block creates its own Bloom filter, a worker picks an entry from one of the input matrix and consults the Bloom filter on the other input. If no match is detected from the Bloom filter, the join execution continues to the next entry; otherwise, a nested-loop join is conducted on the input matrices to generate the join output.

Cost Model for Communications.

MatRel extends MatFast's [1] matrix data partitioner, and it naturally supports three matrix data partitioning schemes: *Row*("*r*"), *Column*("*c*"), and *Broadcast*("*b*"). The Broadcast scheme is only used for sharing a matrix of low dimensions, e.g., a single vector. Different matrix data partitioning schemes lead to significantly different communication overhead for join executions. Therefore, we introduce a cost model to evaluate the communication costs of various partitioning schemes for join operations on matrix data.

We design a communication cost model based on the join predicates. For cross-product, we have

$$C_{comm}(\mathbf{A} \otimes \mathbf{B}) = \begin{cases} 0, & \text{if } s_{A/B} = b. \\ (N-1)\min\{|\mathbf{A}|, |\mathbf{B}|\}, & \text{otherwise.} \end{cases}$$

where s_A and s_B are the partitioning scheme of input matrix $\mathbf{A}$ and $\mathbf{B}$, and N is the number of workers in the cluster. $|\mathbf{A}|$ refers to the size of the Matrix $\mathbf{A}$, i.e., $|\mathbf{A}| = mn$ if $\mathbf{A}$ is an m-by-n dense matrix; and it means $\texttt{nnz}(\mathbf{A})$ if $\mathbf{A}$ is sparse. The communication cost is 0 if one of the inputs has been broadcast to every worker in the cluster. This only applies to the case when either $\mathbf{A}$ or $\mathbf{B}$ is a matrix of tiny dimensions. When $\mathbf{A}$ and $\mathbf{B}$ are partitioned in Row/Column scheme, each partition of the smaller matrix has to be mapped to every worker. Therefore, it incurs a communication cost of $(N-1)\min\{|\mathbf{A}|, |\mathbf{B}|\}$.

For direct overlay, we have the following cost function,

$$C_{comm}(\mathbf{A} \bowtie_{\gamma,f} \mathbf{B}) = \begin{cases} \frac{N-1}{N}\min\{|\mathbf{A}|, |\mathbf{B}|\}, & \text{if } (s_A, s_B) = \\ & (r,c) \text{ or } (c,r). \\ 0, & \text{otherwise.} \end{cases}$$

It is clear that the direct overlay operation induces 0 communication overhead if one of the inputs is broadcast to all the workers in the cluster. Furthermore, there is no communication when both inputs are partitioned using the same scheme. Each worker receives matrix blocks with the same row/column block IDs from both inputs, and the join predicates are naturally satisfied. The communication overhead is only incurred when one of join operands is partitioned in Row scheme and the other is partitioned in Column scheme. To mitigate this partitioning scheme incompatibility, MatRel repartitions the smaller matrix using the same data partitioning scheme from the larger matrix.

For transpose overlay, we have a similar cost function,

$$C_{comm}(\mathbf{A} \bowtie_{\gamma,f} \mathbf{B}) = \begin{cases} \frac{N-1}{N} \min\{|\mathbf{A}|, |\mathbf{B}|\}, & \text{if } (s_A, s_B) = \\ & (r, r) \text{ or } (c, c). \\ 0, & \text{otherwise.} \end{cases}$$

The cost model for a transpose overlay is very similar to that of the direct overlay case. The difference is that the communication overhead is introduced when two matrices are partitioned using the same scheme.

Table 4.1 summarizes the communication costs for joins on a single dimension. If any input matrix is broadcast to all the workers, there is no communication cost. We omit this case in Table 4.1. Let us focus on the case when $\gamma =$ "$RID_A = RID_B$" and $(s_A, s_B) = (r, c)$. Figures 4.5 illustrates that Matrix $\mathbf{A}$ is partitioned in Row scheme and Matrix $\mathbf{B}$ is partitioned in Column scheme. Suppose we have 3 workers in the cluster, i.e., $N = 3$. Each square in the matrix is a block partition. The blocks with the same color are stored on the same worker. The join predicate requires that the entries sharing the same RID's are joined together. There are two possible execution strategies based on the partitioning schemes of $\mathbf{A}$ and $\mathbf{B}$. Strategy I sends the blocks with the same RID from $\mathbf{A}$ to the workers which hold the joining blocks from $\mathbf{B}$. For example, Worker W_1 sends all the blocks to Worker W_2 and W_3, since both W_2 and W_3 hold some joining blocks from $\mathbf{B}$. Both W_2 and W_3 send the blocks in a similar

manner. Each worker needs to send the blocks $(N-1)$ times to all the other workers. Thus, this strategy induces $(N-1)|\mathbf{A}|$ communication cost. Strategy II adopts a different policy by sending different blocks from $\mathbf{B}$ to the same worker that holds the blocks with the same RID from $\mathbf{A}$. For example, the blocks with diagonal lines from W_2 and W_3 are sent to W_1 to satisfy the join predicate. As illustrated in Figure 4.5, all the blocks with diagonal lines introduce a communication cost of $\frac{N-1}{N}|\mathbf{B}|$. Thus, the best communication cost is $\min\{(N-1)|\mathbf{A}|, \frac{N-1}{N}|\mathbf{B}|\}$ when $\gamma =$ "$RID_A = RID_B$" and $(s_A, s_B) = (r, c)$. The remaining entries in Table 4.1 can be computed with similar strategies. Notice, the diagonal of Table 4.1 contains all 0's. This is because the partitioning schemes of both matrices happen to match the join predicate.

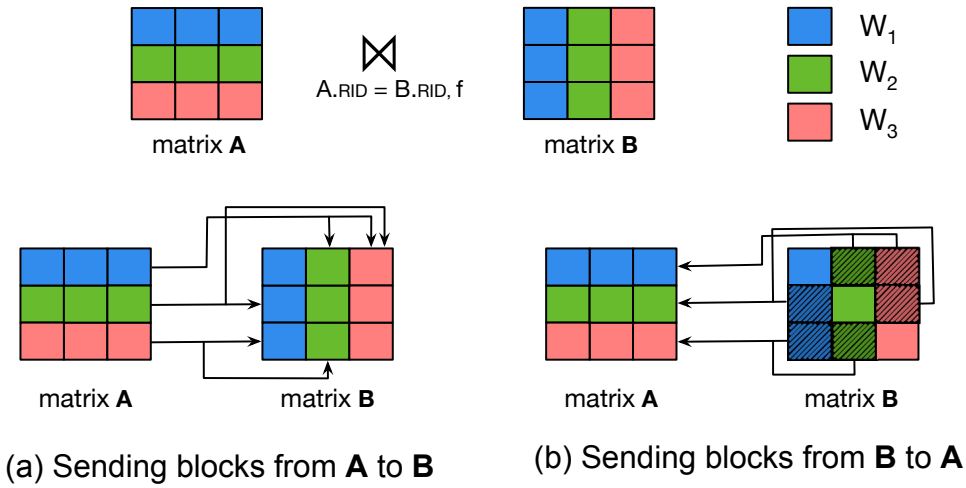


Figure 4.5.: Communication cost for D2D join.

Table 4.1.: Communication cost of different D2D join predicates

	(s_A, s_B)			
γ	(r, r)	(r, c)	(c, r)	(c, c)
$RID_A = RID_B$	0	$\min\{(N-1)\lvert\mathbf{A}\rvert, \frac{N-1}{N}\lvert\mathbf{B}\rvert\}$	$\min\{\frac{N-1}{N}\lvert\mathbf{A}\rvert, (N-1)\lvert\mathbf{B}\rvert\}$	$(N-1)\min\{\lvert\mathbf{A}\rvert, \lvert\mathbf{B}\rvert\}$
$RID_A = CID_B$	$\min\{(N-1)\lvert\mathbf{A}\rvert, \frac{N-1}{N}\lvert\mathbf{B}\rvert\}$	0	$(N-1)\min\{\lvert\mathbf{A}\rvert, \lvert\mathbf{B}\rvert\}$	$\min\{\frac{N-1}{N}\lvert\mathbf{A}\rvert, (N-1)\lvert\mathbf{B}\rvert\}$
$CID_A = RID_B$	$\min\{\frac{N-1}{N}\lvert\mathbf{A}\rvert, (N-1)\lvert\mathbf{B}\rvert\}$	$(N-1)\min\{\lvert\mathbf{A}\rvert, \lvert\mathbf{B}\rvert\}$	0	$\min\{(N-1)\lvert\mathbf{A}\rvert, \frac{N-1}{N}\lvert\mathbf{B}\rvert\}$
$CID_A = CID_B$	$(N-1)\min\{\lvert\mathbf{A}\rvert, \lvert\mathbf{B}\rvert\}$	$\min\{\frac{N-1}{N}\lvert\mathbf{A}\rvert, (N-1)\lvert\mathbf{B}\rvert\}$	$\min\{(N-1)\lvert\mathbf{A}\rvert, \frac{N-1}{N}\lvert\mathbf{B}\rvert\}$	0

A join on entries has an identical cost function as a cross-product. It is clear that the join execution incurs 0 communication cost if either $\mathbf{A}$ or $\mathbf{B}$ is partitioned in Broadcast scheme. For any other combination of partitioning schemes of $\mathbf{A}$ and $\mathbf{B}$, a worker needs to broadcast the smaller of the inputs to all the other workers. Hence, the communication cost is $(N-1)\min\{|\mathbf{A}|, |\mathbf{B}|\}$.

For a join on a single dimension and entries, Table 4.2 gives the communication costs for the various schemes. We use η to denote the selectivity of entries that could match the row/column dimensions from the other matrix. Let us examine the cost function for $\gamma =$ "$RID_A = val_B$". When $(s_A, s_B) = (r, r)$, there exist two strategies to evaluate the join. Strategy I requires each worker to send its own matrix blocks of $\mathbf{A}$ to the other $(N-1)$ workers, since any worker may hold a block from $\mathbf{B}$ that has matched entries with the row dimension. Broadcasting $\mathbf{A}$ incurs a communication cost of $(N-1)|\mathbf{A}|$. Strategy II sends the matched entries from $\mathbf{B}$ to each corresponding worker that holds the blocks of $\mathbf{A}$. The selectivity indicates a total amount of $\eta_B|\mathbf{B}|$ of matrix data is transferred. Thus, the communication cost would be $\min\{(N-1)|\mathbf{A}|, \eta_B|\mathbf{B}|\}$. For $(s_A, s_B) = (c, r)$, the only difference is that the selected matching entries from $\mathbf{B}$ are sent to all the workers in the cluster, since $\mathbf{A}$ is partitioned in Column scheme. The remaining entries in Table 4.2 can be computed in a similar manner.

Table 4.2.: Communication cost of different D2V and V2D join predicates

	(s_A, s_B)			
γ	(r, r)	(r, c)	(c, r)	(c, c)
$RID_A = val_B$	$\min\{(N-1)\|\mathbf{A}\|, \eta_B\|\mathbf{B}\|\}$	$\min\{(N-1)\|\mathbf{A}\|, \eta_B\|\mathbf{B}\|\}$	$\min\{(N-1)\|\mathbf{A}\|, N\eta_B\|\mathbf{B}\|\}$	$\min\{(N-1)\|\mathbf{A}\|, N\eta_B\|\mathbf{B}\|\}$
$CID_A = val_B$	$\min\{(N-1)\|\mathbf{A}\|, N\eta_B\|\mathbf{B}\|\}$	$\min\{(N-1)\|\mathbf{A}\|, N\eta_B\|\mathbf{B}\|\}$	$\min\{(N-1)\|\mathbf{A}\|, \eta_B\|\mathbf{B}\|\}$	$\min\{(N-1)\|\mathbf{A}\|, \eta_B\|\mathbf{B}\|\}$
$val_A = RID_B$	$\min\{\eta_A\|\mathbf{A}\|, (N-1)\|\mathbf{B}\|\}$	$\min\{N\eta_A\|\mathbf{A}\|, (N-1)\|\mathbf{B}\|\}$	$\min\{\eta_A\|\mathbf{A}\|, (N-1)\|\mathbf{B}\|\}$	$\min\{N\eta_A\|\mathbf{A}\|, (N-1)\|\mathbf{B}\|\}$
$val_A = CID_B$	$\min\{N\eta_A\|\mathbf{A}\|, (N-1)\|\mathbf{B}\|\}$	$\min\{\eta_A\|\mathbf{A}\|, (N-1)\|\mathbf{B}\|\}$	$\min\{N\eta_A\|\mathbf{A}\|, (N-1)\|\mathbf{B}\|\}$	$\min\{\eta_A\|\mathbf{A}\|, (N-1)\|\mathbf{B}\|\}$

Algorithm for Partitioning Scheme Assignment of Joins.

Before we delve into the algorithm of partitioning scheme assignment for joins, we first discuss the cost function for converting distributed matrix data from one partitioning scheme to another. Table 4.3 illustrates the conversion costs, where s_A is the partition scheme of an input, and s'_A is the scheme for an output. ξ denotes the case when the input data is randomly partitioned among all the workers in the cluster, e.g., round-robin. It introduces the communication cost of $|\mathbf{A}|$ when re-partitioning Matrix $\mathbf{A}$ from round-robin to Row/Column scheme. Broadcast scheme is more expensive and it costs $N|\mathbf{A}|$, where each worker has a copy of all the matrix data.

Table 4.3.: Communication cost of converting partition schemes

	s'_A		
s_A	r	c	b
r	0	$\frac{N-1}{N}\|\mathbf{A}\|$	$(N-1)\|\mathbf{A}\|$
c	$\frac{N-1}{N}\|\mathbf{A}\|$	0	$(N-1)\|\mathbf{A}\|$
b	0	0	0
ξ	$\|\mathbf{A}\|$	$\|\mathbf{A}\|$	$N\|\mathbf{A}\|$

Given a join operation and input matrices $\mathbf{A}$ and $\mathbf{B}$, MatRel's data partitioner computes the best partition schemes for $\mathbf{A}$ and $\mathbf{B}$ by optimizing the following,

$$(s'_A, s'_B) \leftarrow \underset{(s'_A, s'_B)}{\arg\min} \{C_{comm}(\mathbf{A} \bowtie_{\gamma,f} \mathbf{B}, s'_A, s'_B) + C_{vt}(\mathbf{A}, s_A \rightarrow s'_A) + C_{vt}(\mathbf{B}, s_B \rightarrow s'_B)\}.$$

Function $C_{comm}(\mathbf{A} \bowtie_{\gamma,f} \mathbf{B}, s'_A, s'_B)$ computes the communication cost of the join according to our cost model when $\mathbf{A}$ is partitioned in Scheme s_A and $\mathbf{B}$ in Scheme s_B. Function $C_{vt}(\mathbf{A}, s_A \rightarrow s'_A)$ computes the communication cost when converting $\mathbf{A}$ from Scheme s_A to s'_A. Essentially, the data partitioner adopts a grid-search strategy among all the possible combinations of partition schemes on the input matrices, and outputs the cheapest schemes for $\mathbf{A}$ and $\mathbf{B}$.

4.4 System Implementation

After query execution plan and matrix data partitioning schemes are generated, each worker conducts matrix and relational computations locally. MatRel leverages block matrices as a basic unit for storing and manipulating matrix data in the distributed memory. We discuss briefly the system implementation on top of Spark SQL.

4.4.1 Physical Storage of Matrix Joins (Tensor)

We partition a matrix into smaller square blocks to store and manipulate matrix data in the distributed memory. A matrix block is the basic unit for storage and computation. Every matrix block consists of two parts: a block ID and matrix data. A block ID is an ordered pair, i.e., (row-block-ID, column-block-ID). The matrix data field is a quadruple, ⟨matrix format, number of rows, number of columns, data storage⟩. A local matrix block supports both dense and sparse matrix storage formats. For the sparse format, the nonzero entries are stored in Compressed Sparse Column (CSC), and Compressed Sparse Row (CSR) format. Refer to Section 3.4.1 for more details.

A join may produce a normal matrix or a higher-dimensional tensor, e.g., a 3rd-order tensor for joins on a single dimension. MatRel utilizes block matrices to manipulate higher-dimensional tensors as well. Given a 3rd-order tensor, the schema of the tensor is represented as `(D1, D2, D3, val)`, where `D1`, `D2`, and `D3` denote the three dimensions. To leverage the block matrix storage, MatRel extends the block ID component to higher dimensions, e.g., (D1, D2-block-ID, D3-block-ID), where D1 records the exact dimension value, and the remaining record the matrix block IDs for the other dimensions. Figure 4.6 illustrates the storage layout of a 3rd-order tensor in terms of matrix blocks. There is freedom on choosing which dimension of the tensor to serve as D1, D2, or D3. Usually there exits an aggregation on a certain dimension following the join. A heuristic is to choose a non-aggregated

dimension as D1 in a physical block storage. It is beneficial that each worker only needs to execute the aggregation locally without further communication. Figure 4.6 demonstrates 2 possible tensor layouts on dimension D1 and D3 respectively. If a subsequent aggregation is performed on dimension D2 or D3, MatRel stores the join result in block matrices with respect to dimension D1.

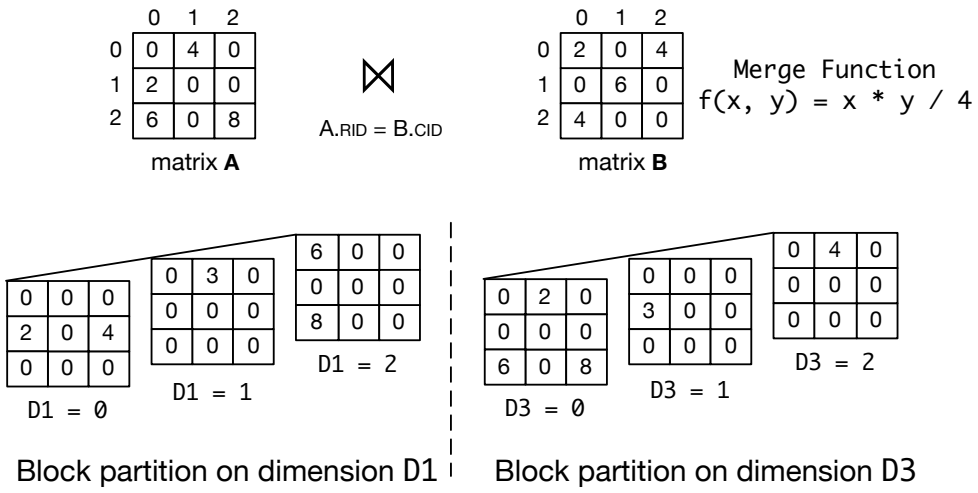


Figure 4.6.: Block tensor storage.

4.4.2 System Design and Implementation

MatRel is implemented as a library in Apache Spark. It extends Spark SQL and its Catalyst optimizer to work seamlessly with matrix/tensor data. We provide Scala API for conducting relational query processing on distributed matrix data. It uses a DataFrame [65] to represent distributed matrix blocks. Each row in a DataFrame has the schema `(RowBlkID, ColBlkID, Matrix)`, where both IDs takes a long integer type, and `Matrix` is a user-defined type that encodes the quadruple shown in Figure 3.5. For logical optimization, we extend the Catalyst optimizer with all the transformation rules for optimizing the relational operations on matrix data. Once an optimized logical plan is produced, MatRel generates one or more physical plans. By leveraging the cost model for communication, the matrix data partitioner selects

the partitioning schemes for input matrices, that incurs minimum communication overhead. Finally, the optimized physical plan is realized by RDD's transformation operations, e.g., `map`, `flatMap`, `zipPartitions`, and `reduceByKey`, etc. The RDD partitioner class is extended with three partitioning schemes for distributed matrices, i.e., Row, Column, and Broadcast. Spark's fault tolerance mechanism applies naturally to MatRel. In case of node failure, the lost node is evicted and a standby node is chosen to recover the lost one. An open-source version of MatRel is available on GitHub[2].

4.5 Related Work

We review related work of relational and matrix query processing and optimizations on matrix data.

Matrix Query Processing on Matrices

There has been a long history of research on conducting efficient matrix computations in the high-performance computing (HPC) community. Existing libraries provide efficient matrix operators, e.g., BLAS [52] and LAPACK [53] (and its distributed variant ScaLAPACK [60]). However, they lack support for sparse matrices, and are prone to machine failures.

Recently, many systems have been proposed to support efficient matrix computations using Hadoop [30] and Spark [51]. These systems provide an easy-to-use interface on multiple matrix operations, e.g., HAMA [34], Mahout [35], MLI [39], MadLINQ [56], Cumulon [69], SystemML [36, 57, 58, 70, 71], DMac [41], and MatFast [1]. Each system provides various optimization strategies for reducing memory consumption, computation and communication overhead. For example, SystemML [58] adopts column encoding schemes and operations over compressed matrices to mitigate memory overhead when the original matrices are too large to fit in the available compute resources. MatFast [1] adopts a sampling-based approach

[2] `https://github.com/yuyongyang800/SparkDistributedMatrix/tree/spark-2.1-dev`

to estimate the sparsity of matrix chain multiplications, and organizes distributed matrix partitions in a communication-efficient manner by leveraging matrix data dependencies. However, these systems focus on *matrix-only* operations on big matrix data. None provides *full-fledged* relational query processing on matrix data. MatRel is built on MatFast, and provides optimized relational query processing on the distributed matrix data. MATLANG [72] examines matrix manipulation from the view of expressive power of database query language. It confirms the matrix operators provided by these systems are adequate for a wide range of applications.

Matrix Query Processing on Relations

A lot of work consider the problem of providing ML over relational data for specific ML algorithms. The assumption is the input dataset can be viewed as a join result on multiple tables. The so-called "factorized machine learning" tries to learn a model based on the de-normalized tables, without performing the join operation explicitly. For instance, [73] aims at optimizing the linear regression model over factorized join tables. [74] shows how to learn a generalized linear model over factorized join tables. Furthermore, [75] discusses the cases when it is safe to conduct feature selection by avoiding key-foreign key joins to obtain features from all base tables. More recently, Morpheus [76] is proposed to conduct general linear algebra operations over factorized tables for popular ML algorithms. However, these systems rely on the assumption that the input matrix is obtained from *joins on multiple tables*, which is *not* always the case for real-world applications, e.g., recommender systems.

Relational Query Processing on Matrices

The idea of building a universal array DBMS on multidimensional arrays for scientific and numerical applications has been explored for a long time. One of the most notable efforts is Rasdaman [77].

Recently, there are several systems built from scratch with native support for linear algebra, such as SciDB [59, 78, 79] and TensorFlow [80]. TensorFlow has limited support for distributed computation. The user has to manually map computation and data to each worker, since Tensorflow does not offer automatic work assign-

ment [81]. While Tensorflow is mainly designed for neural network models, it lacks well-defined relational operations on a matrix (tensor). SciDB supports the array data model, and adopts a share-nothing massively parallel processing (MPP) architecture. SciDB provides a rich set of relational operations on the array data. However, it treats each array operator individually without tuning for a series of array operators. ChronosDB [82] is an array database, specially designed for processing raster data. It delegates portions of raster data processing to feature-rich and highly optimized command line tools. It is different from MatRel, since the operators are only optimized for satellite images in certain formats, while MatRel provides optimizations on generic matrix data. Furthermore, ChronosDB does not provide holistic optimizations over a series of operation on raster data.

The MADlib project [83] built analytics, including linear algebra functionality, on top of a database system. MADlib shows the potential of high-performance of linear algebra on top of a relational database. Recent extension on SimSQL [84] discusses the possibility of making a small set of changes to SQL for enabling a distributed, relational database engine to become a high-performance platform for distributed linear algebra. However, the extension does not cover the optimization on a series of mixed relational and matrix operations. Our work explores the potential to optimize the query execution pipeline by pushing relational operators below matrix operators, and computes costs for different plans. MatRel's query optimizer also takes into account the optimized data layout of join operands for communication-efficient executions.

4.6 Performance Evaluation

We study the performance of the optimized execution plans by conducting different relational operations on matrix data from various ML applications. The performance is measured by the average execution time.

4.6.1 Experiment Setup

The experiments are conducted on an HP DL360G9 cluster with Intel Xeon E5-2660 realized over 6 nodes. The cluster uses Cloudera 5.9 consisting of Spark 2.1 as a computational framework and Hadoop HDFS as a distributed file system. Each node has 16 cores, 32 GB of RAM, and 400 GB of local storage. The total HDFS size is 1 Terabyte. Spark was configured with 5 executors, 16 cores/executor, 16 GB driver memory, and 24 GB executor memory.

Comparison Across Different Platforms

Platforms Tested. The platforms we evaluated are:

(1) MatRel. This is our implementation on Spark 2.1. All computations are written in Scala using the extended DataFrame API. The experiments are conducted on MatRel and MatRel(w/o-opt), where the optimizations are turned on and off respectively.

(2) Spark MLlib. This is the built-in Spark library, `mllib.linalg`. This is run on Spark V2.1 in the cluster mode. All computations are written in Scala.

(3) SystemML. This is SystemML V0.13.1, which provides the option to run on top of Apache Spark. All computations are written in SystemML's DML programming language.

(4) SciDB. This is SciDB V15.12.1. All computations are written in SciDB's AQL language that is similar to SQL. Dense matrix computations are conducted with `gemm()` API, and sparse matrix computations are with `spgemm()` API.

Datasets

Our experiments are performed on both real-world and synthetic datasets. The three real-world datasets are: soc-pokec[3], cit-Patents[3], and LiveJournal[3]. All these datasets are sparse matrices and their statistics are shown in Table 4.4.

[3] `https://snap.stanford.edu/data/`

Table 4.4.: Statistics of the social network datasets

Graph	**#nodes**	**#edges**
soc-pokec	1,632,803	30,622,564
cit-Patents	3,774,768	16,518,978
LiveJournal	4,847,571	68,993,773

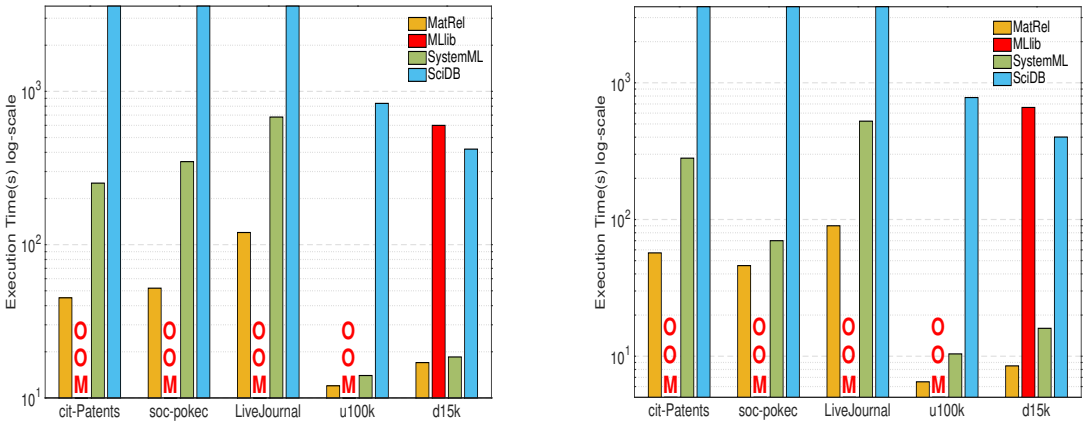
Furthermore, we generate extra smaller datasets, e.g., `u100k` and `d15k`. The first letter denotes the matrix type, i.e., "`u`" for a sparse matrix with a uniform distribution of the nonzero entries, and "`d`" for a dense matrix. The trailing number indicates the dimension of the square matrix. For example, `u100k` is a 100K-by-100K sparse matrix, where nonzero entries are uniformly distributed. `d15k` is a 15K-by-15K dense matrix.

In our experiments, we conduct several representative computations. We manually kill the job if it fails to finish within 1 hour, or 3600s.

4.6.2 Aggregation on a Gram Matrix

A Gram matrix is computed as the inner-products of a set of vectors. It is commonly used in ML applications to compute the kernel functions and covariance matrices. Given a matrix $\mathbf{X}$ to store the input vectors, the Gram matrix can be computed as $\mathbf{G} = \mathbf{X}^T \times \mathbf{X}$. We consider two different aggregations on the Gram matrix, i.e., summation along the row dimension $\Gamma_{\texttt{sum},r}(\mathbf{G})$, and trace computation $\Gamma_{\texttt{sum},d}(\mathbf{G})$.

Figure 4.7a illustrates the execution time for different systems when performing $\Gamma_{\texttt{sum},r}(\mathbf{X}^T \times \mathbf{X})$. For all the sparse matrices, Spark MLlib throws out-of-memory (OOM) exceptions, due to the fact that MLlib converts a sparse matrix to the dense format for matrix-matrix multiplications. Even for the smallest 100K-by-100K sparse matrix with the sparsity of 10^{-5}, it requires about 160 GB memory to store the dense matrices, which exceeds the hardware limit of our cluster. Without pushing the aggregation below the matrix-matrix multiplications, SciDB runs over 1 hour to compute



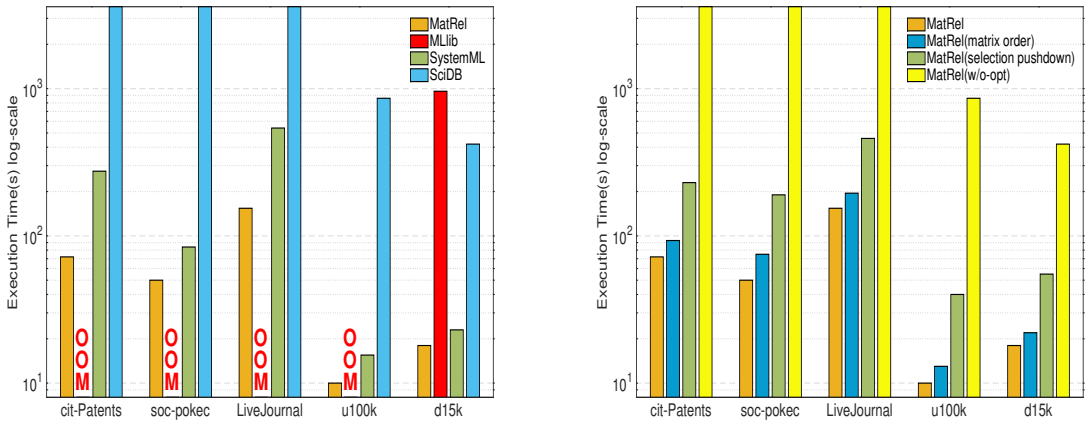
(a) Execution time for $\Gamma_{\texttt{sum},r}(\mathbf{G})$

(b) Execution time for $\Gamma_{\texttt{sum},d}(\mathbf{G})$

Figure 4.7.: Sum aggregation over matrix-matrix multiplications.

the product of two matrices before the aggregation. On the other hand, both MatRel and SystemML adopt the similar rewrite rule to push `sum` aggregation below the matrix-matrix multiplications. MatRel and SystemML spend 120s and 680s for the computation on LiveJournal dataset respectively. The extra performance gain for MatRel comes from the fact that MatRel is implemented on Spark Dataset API while SystemML is run on RDDs directly. A Dataset makes extra effort for efficient data compression, serialization, and de-serialization. For `u100k` dataset, SciDB spends about 800s to compute the sparse matrix multiplication, since SciDB is not optimized for sparse matrix computations. For `d15k` dataset, all systems finish the job within the time budget.

Figure 4.7b shows the execution time for different systems when performing $\Gamma_{\texttt{sum},d}(\mathbf{X}^T \times \mathbf{X})$. The trace computation leverages the rule to rewrite matrix-matrix multiplications in terms of matrix element-wise multiplications. MatRel's optimizer detects the two inputs are table scans on the same matrix after query rewrite. It generates the code to compute the matrix element-wise multiplication on a single matrix without duplicate table scans. For the LiveJournal dataset, it takes about 90s



(a) Execution time for LR (b) Effects of various optimizations in MatRel

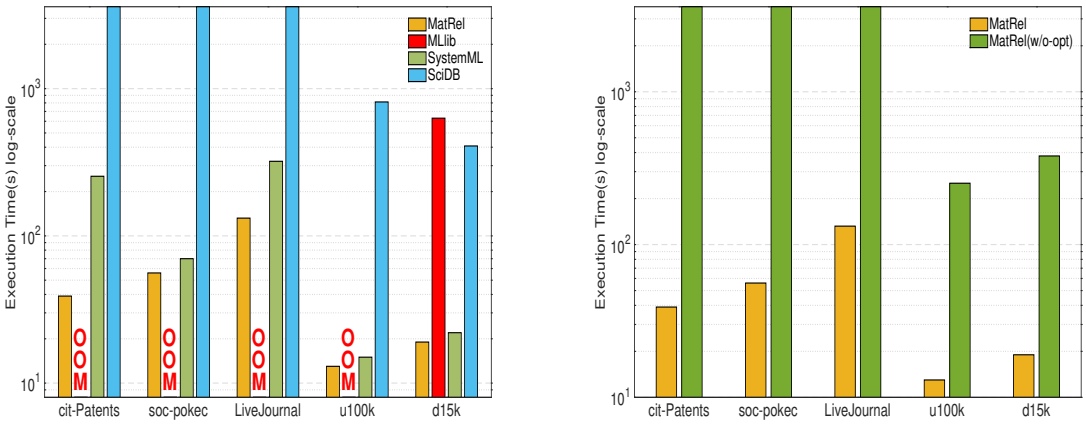
Figure 4.8.: Selecting a row from linear regression.

for MatRel and 525s for SystemML to complete the query execution. Other systems spend similar amount of time as the previous query since they fail to leverage the more efficient query rewrite rule.

4.6.3 Selection over Matrix Data

Least squares linear regression (LR) is a popular ML model for classification [23]. The input is a feature matrix $\mathbf{X}$ and a label vector $\mathbf{y}$. Each label y_i can be viewed as a linear combination of the feature vector $\mathbf{x}_i^T$, i.e., $y_i \approx \mathbf{x}_i^T \times \mathbf{b} + \varepsilon_i$, where $\mathbf{b}$ is the vector of regression coefficients, and ε_i is the error term. The most common estimator for $\mathbf{b}$ is the least squares estimator $\hat{\mathbf{b}} = (\mathbf{X}^T \times \mathbf{X})^{-1} \times \mathbf{X}^T \times \mathbf{y}$.

Figure 4.8a illustrates the execution time for selecting a row of $\hat{\mathbf{b}}$. For an efficient evaluation of the coefficient vector $\hat{\mathbf{b}}$, a good execution plan should compute $\mathbf{X}^T \times \mathbf{y}$ before multiplying with $(\mathbf{X}^T \times \mathbf{X})^{-1}$, since multiplying $\mathbf{X}^T$ with $\mathbf{y}$ results in a lower dimension matrix. MatRel adopts this optimized evaluation plan. It also selects a single row on $(\mathbf{X}^T \times \mathbf{X})^{-1}$. Spark MLlib throws OOM exceptions for all the sparse matrices due to the inefficient implementation on matrix multiplications. SystemML



(a) $\sigma_{RID=1 \wedge CID=1}(\mathbf{G})$

(b) Pushing selection below matrix multiplications

Figure 4.9.: Selecting an entry from a Gram matrix.

adopts a similar strategy for selection pushdown. However, the implementation on RDD prohibits SystemML from obtaining the better performance achieved by MatRel, which implements the operations on Datasets. SciDB does not generate an efficient execution plan for LR computation, and performs the selection only *after* the complete evaluation of $\hat{\mathbf{b}}$. Figure 4.8b demonstrates the different effects of optimizations that MatRel has adopted. Especially, we manually turn off the optimizations of order selection on matrix chain multiplications, and relational selection pushdown below matrix multiplications, e.g., MatRel (matrix order) means turning on the optimization of order selection on matrix chain multiplications only.

Figure 4.9a demonstrates the execution time for evaluating a selection on a matrix entry of the Gram matrix. All the systems conduct Gram matrix computation in the same manner. With selection pushdown, MatRel and SystemML are able to finish the computation in 39s and 254s for the cit-Patents dataset. When selecting (i, j)-th entry on $\mathbf{X}^T \times \mathbf{X}$, MatRel's optimizer rewrites the query plan to $(\sigma_{CID=i}(\mathbf{X}))^T \times \sigma_{CID=j}(\mathbf{X})$. The rewritten query plan avoids evaluation of a matrix transpose by

a vector transpose. SciDB cannot finish the evaluation of the matrix multiplication $\mathbf{X}^T \times \mathbf{X}$ in 3600s. Figure 4.9b illustrates the effect of selection pushdown below matrix multiplications on MatRel.

4.6.4 Cross-product

The Kronecker product is a generalization of outer-product from vectors to matrices. It is widely used in tensor decomposition and applications [31]. Formally, given an m-by-n matrix $\mathbf{A}$ and a p-by-q matrix $\mathbf{B}$, the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ is the mp-by-nq block matrix,

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11} * \mathbf{B} & \dots & a_{1n} * \mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1} * \mathbf{B} & \dots & a_{mn} * \mathbf{B} \end{bmatrix}.$$

The Kronecker product is *essentially* the cross-product between matrix $\mathbf{A}$ and $\mathbf{B}$ with the merge function of $f(x, y) = x * y$. We compare MatRel and SciDB for the performance of the Kronecker product computation, since all the other systems do not support joins. The Kronecker product is an expensive operation that consumes lots of resources. Therefore, we generate another 25K-by-25K sparse matrix `u25k`, where nonzero entries are uniformly distributed with sparsity of 10^{-6}. Table 4.5 demonstrates the execution time for different systems on various cross-product tasks.

Table 4.5.: The Kronecker product on different systems

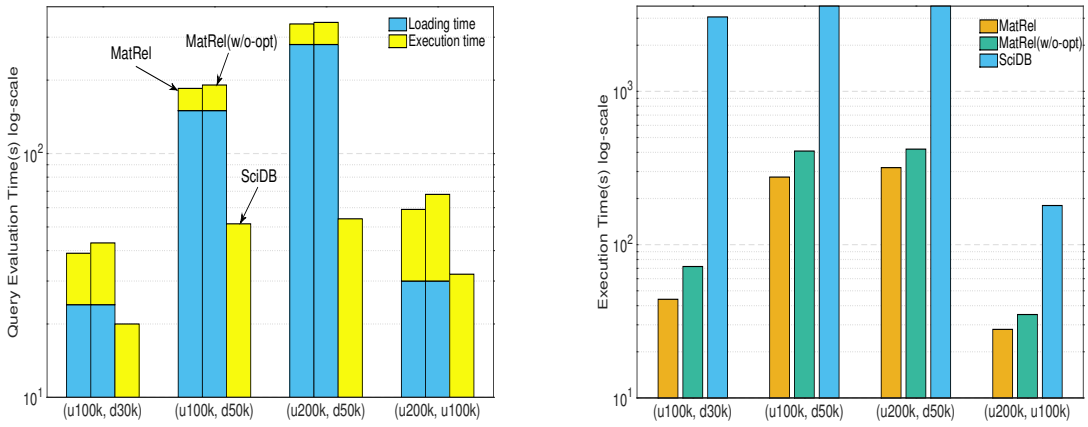
Dataset	**MatRel**	**MatRel(w/o)**	**SciDB**
`u25k` $\otimes$ `d15k`	294s	$>$ 1h	$>$ 1h
`u25k` $\otimes$ `u25k`	44s	$>$ 1h	$>$ 1h
`d15k` $\otimes$ `u25k`	312s	$>$ 1h	$>$ 1h
`d15k` $\otimes$ `d15k`	OOM	OOM	NSLOD

Both MatRel(w/o-opt) and SciDB first conduct the cross-product computation on each pair of matching entries. Then, they evaluate the merge function on the matched entries. On the other hand, MatRel's optimizer identifies that merge function $f(x, y)$ has the zero-preserving property. Thus, MatRel only computes the cross-product using the nonzero entries from the sparse input. MatRel spends about 294s to finish the evaluation of the Kronecker product between a dense matrix and a sparse one, while MatRel(w/o-opt) and SciDB cannot finish the job within 1 hour. MatRel spends only 44s for computing the Kronecker product between two sparse matrices. When both inputs are dense matrices, no optimizations can be applied, and no system can finish the job successfully. There are totally $15^4 \times 10^{12}$ entries when computing the Kronecker product between two 15K-by-15K dense matrices, which costs about 4×10^5 TB. This is far beyond the total amount of memory of our cluster, or even the disk space. Thus, MatRel throws OOM exceptions after 5 minutes, and SciDB throws no space left on device (NSLOD) errors after 3 hours.

4.6.5 Join on dimensions

Many real-world applications rely on joins on dimensions of big matrices, e.g., raster data overlay analysis. We examine two different join predicates: equal-join on two dimensions, and equal-join on a single dimension. Among all the systems we compare with, only SciDB provides similar functionality with `join()` and `cross_join()`. The `join()` operator combines the entires of two input matrices at matching dimension values. The `cross_join()` operator computes the cross-product of the two input matrices, and applies equality predicates to pairs of dimensions.

We conduct experiments on two different types of joins: direct overlay and transpose overlay. The `join()` operator implements direct overlay exactly. The `cross_join()` operator is leveraged to evaluate the transpose overlay. Figure 4.10a illustrates the execution time of MatRel and SciDB on various combinations of inputs. `d30k` and `d50k` are two dense square matrices with dimensions 30K and 50K respectively. Both



(a) Direct overlay

(b) Transpose overlay

Figure 4.10.: Execution time for join on two dimensions.

`u100k` and `u200k` are sparse matrices with the sparsity of 10^{-5}. Thanks to the Multidimensional Array Clustering (MAC) technique, SciDB stores matrix partitions in a way that the matrix chunks are aligned along the dimensions. This data layout is preferred for evaluating a direct overlay, since the join can be performed locally on the matched chunks. On the other hand, MatRel does not make any assumptions on data locations when loading data from HDFS. It needs to shuffle the matrix blocks to fulfill the requirement of Row/Column partitioners for join executions. Figure 4.10a shows a detailed decomposition of the execution time of MatRel. The execution time is divided into two parts, the matrix data loading time and query execution time. For SciDB, there is only execution time since the data is already partitioned. The *actual* execution time of MatRel is similar to that of SciDB. The MatRel(w/o-opt) uses the default hash partitioner of Spark, which has a similar performance to MatRel for direct overlay queries.

Figure 4.10b shows the performance comparison for the transpose overlay queries. For this type of query, SciDB no longer benefits from the MAC technique, since the matrix chunks have to be transferred to different workers to facilitate the query

execution. For example, SciDB spends about 51min to conduct the transpose overlay query when joining `u100k` and `d30k`. On the other hand, it takes about 44s for MatRel to finish the same query, which shows a consistent performance as the direct overlay query. When testing on larger datasets, say `u100k` and `d50k`, SciDB spends about 172min to complete, while MatRel only spends 276s.

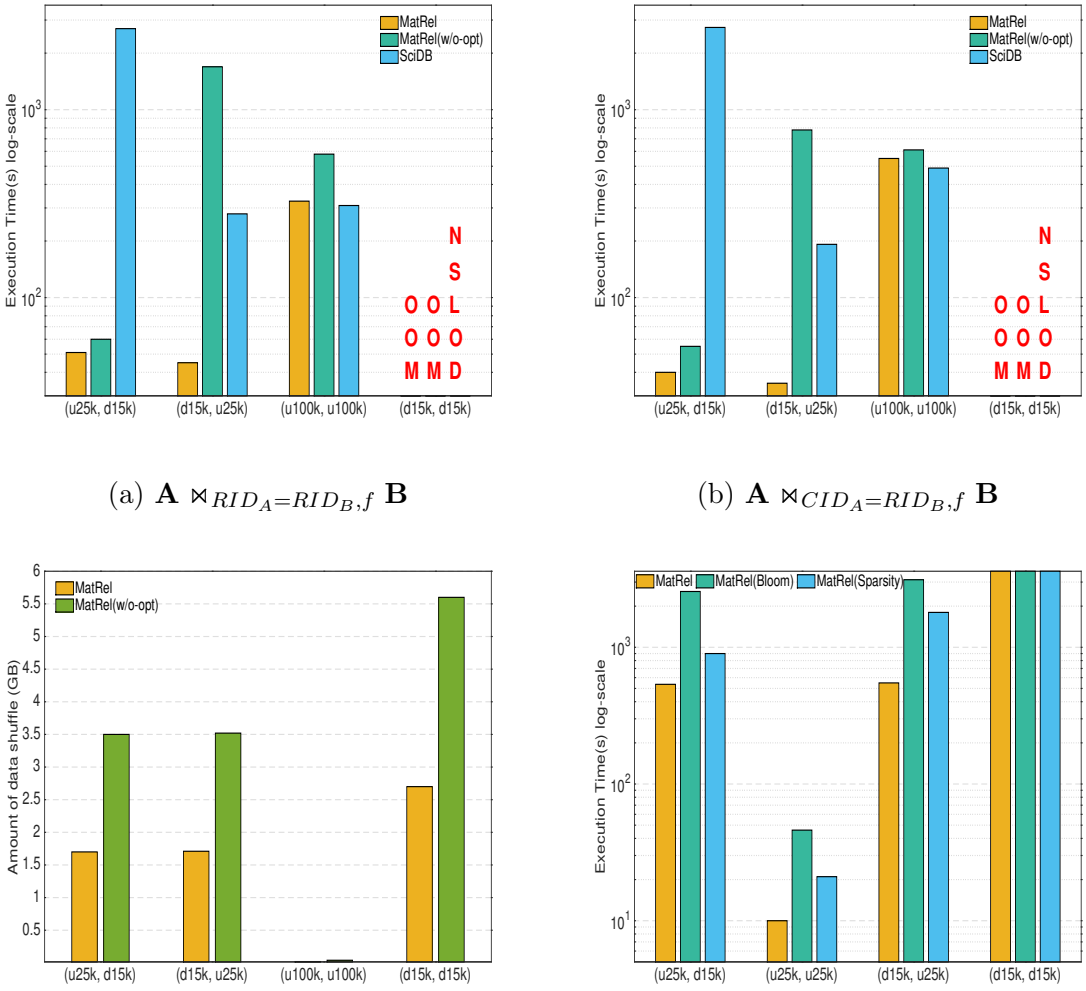
We further perform experiments for joins on a single dimension. Figure 4.11a illustrates the performance comparison for $\mathbf{A} \bowtie_{RID_A=RID_B,f} \mathbf{B}$, where $f(x, y) = x * y$. We use different combinations of input matrices, e.g., (`u25k`, `d15k`). MatRel leverages the sparsity-preserving property of the merge function $f(x, y)$. MatRel spends about 51s, and SciDB spends about 45min when computing the join on $\mathbf{A}$(`u25k`) and $\mathbf{B}$(`d15k`). MatRel(w/o-opt) converts a sparse matrix to the dense format, and conducts the join on the dense block, spending about 60s. SciDB's `cross_join()` implementation is sensitive to the order of arguments, and it incurs huge overhead when the inner join matrix is a dense one. When swapping matrix $\mathbf{A}$ and $\mathbf{B}$, SciDB's performance improves significantly, and it only takes about 279s to complete the join. MatRel(w/o-opt) spends about 1690s to finish the join, since it has to duplicate the outer dense matrix multiple times and converting a sparse matrix block to a dense one for query evaluation. MatRel's optimizer detects both the dimensions and the sparsity of the input matrices, and shuffles the matrix with fewer nonzero entries to meet the join predicate with the other matrix for better performance.

When both inputs are sparse, MatRel and SciDB exhibit similar performance. It takes longer time for MatRel(w/o-opt) to evaluate the query as it does not leverage the sparsity of the matrices. When both inputs are dense matrices, no optimizations can be applied, and no system can finish the job successfully. For `d25k` dataset, each block size is 1K-by-1K and there are 225 blocks in each input. The join generates 1K new blocks for each input block. That results in about 1800 GB, which exceeds the total amount of available memory of our cluster, or even the free disk space. However, MatRel throws OOM exceptions after about 5 minutes, while SciDB throws NSLOD errors after 3 hours.

Figure 4.11b illustrates the performance comparison for the join predicate of "$CID_A = RID_B$". All the systems show similar trends as Figure 4.11a. Figure 4.11c depicts the amounts of data shuffle for MatRel and MatRel(w/o-opt) when the join predicate is "$RID_A = RID_B$". According to our communication cost model, MatRel's optimizer partitions both **A** and **B** using *Row* scheme. MatRel(w/o-opt) relies on Spark's default hash partitioner to distribute the matrix among all the workers. For a load-balanced execution, we need to repartition the matrices to satisfy the join predicate. Figure 4.11c verifies our cost model since MatRel(w/o-opt) shuffles twice as much data as MatRel.

4.6.6 Join on Entries

For all the systems we've compared with, only MatRel supports the joins when the predicate involves matrix entries. We leverage the sparsity-preserving merge function $f(x, y) = x * y$ for this query. MatRel takes advantages of two optimizations for this kind of queries, i.e., identifying sparsity-preserving merge functions, and leveraging a Bloom join on matrix blocks. Figure 4.11d demonstrates the performance of joins on matrix entries of MatRel when turning on and off the optimizations. MatRel(Bloom) leverages a Bloom filter for probing the entries of the inner matrix blocks without exploiting the sparsity of a sparse input. On the other hand, MatRel(sparsity) examines non-empty blocks in the inner matrix when conducting the join. In general, MatRel(sparsity) achieves a better performance than MatRel(Bloom) when there exists a sparse input. When both inputs are dense matrices, the optimization of sparsity-preserving merge function cannot apply. All the three variants of MatRel cannot finish the query within 1 hour.



(a) $\mathbf{A} \bowtie_{RID_A=RID_B,f} \mathbf{B}$

(b) $\mathbf{A} \bowtie_{CID_A=RID_B,f} \mathbf{B}$

(c) Data shuffle

(d) Join on entries

Figure 4.11.: Join on a single dimension and join on entries.

4.6.7 PNMF

PNMF [85] is a popular model for dimension reduction, and it tries to approximate a sparse input matrix $\mathbf{A}$ with two factor matrices $\mathbf{W}$ and $\mathbf{H}$ of low rank k, typically 10 – 200. The computation steps are

$$\mathbf{W} \leftarrow \mathbf{W} * ((\mathbf{A}/(\mathbf{W} \times \mathbf{H})) \times \mathbf{H}^T)/(\mathbf{E} \times \mathbf{H}^T),$$
$$\mathbf{H} \leftarrow \mathbf{H} * (\mathbf{W}^T \times (\mathbf{A}/(\mathbf{W} \times \mathbf{H})))/(\mathbf{W}^T \times \mathbf{E}),$$

where $\mathbf{A}_{m \times n}$ is the input sparse matrix, $\mathbf{W}_{m \times k}$ and $\mathbf{H}_{k \times n}$ are two factor matrices, and $\mathbf{E}$ is an m-by-n matrix, where $E_{ij} = 1$. The updates for $\mathbf{W}$ and $\mathbf{H}$ continue until convergence. The objective function of PNMF is a combination of Euclidean distance and Kullback-Leibler divergence, i.e.,

$$f(\mathbf{W}, \mathbf{H}) = \sum_{i,j} (\mathbf{W} \times \mathbf{H})_{ij} - \sum_{i,j} (\mathbf{A} * \log(\mathbf{W} \times \mathbf{H}))_{ij}$$

where $\log(\mathbf{X})$ computes the logarithm of each entry in $\mathbf{X}$.

The sparsity of each generated sparse matrix is 10^{-3}. Table 4.6 shows the execution time per iteration of PNMF on different systems. MatRel performs the best, followed by SystemML. MLlib has a better performance than SciDB for dataset of small sizes. However, MLlib does not scale for larger datasets, since it does not adopts native matrix multiplications for sparse matrices.

Table 4.6.: PNMF on different systems

Dataset	**MatRel**	**MLlib**	**SystemML**	**SciDB**
`u25k`	13s	61s	23s	62s
`u50k`	28s	372s	37s	260s
`u100k`	52s	2041s	63s	1092s
`u200k`	148s	OOM	160s	$>$ 1h

Notice, there are lots of opportunities when the sparsity-preserving property could be leveraged, e.g., $\mathbf{A}/(\mathbf{W} \times \mathbf{H})$ and $\mathbf{A} * \log(\mathbf{W} \times \mathbf{H})$. Matrix $\mathbf{A}$ is sparse, thus, it

is unnecessary to evaluate the dense intermediate matrix of $\mathbf{W} \times \mathbf{H}$. MatRel only computes the blocks of $\mathbf{W} \times \mathbf{H}$, which map to the corresponding locations of sparse blocks from $\mathbf{A}$. A similar argument also applies to $\mathbf{A} * \log(\mathbf{W} \times \mathbf{H})$. Furthermore, MatRel adopts the rule of aggregation pushdown below matrix multiplications when evaluating $\sum_{i,j}(\mathbf{W} \times \mathbf{H})_{ij}$ of the objective function. MatRel also interprets $\mathbf{E} \times \mathbf{H}^T$ into $\Gamma_{\mathtt{sum},r}(\mathbf{H})$. Therefore, MatRel involves no matrix multiplications for evaluating the PNMF pipeline.

4.7 Concluding Remarks

In this chapter, We have presented MatRel, an in-memory system that enables scalable relational query processing on big matrix data in a distributed setup. MatRel supports common relational operations on big matrix data, e.g., relational selection, projection, aggregation, join. MatRel's query optimizer leverages rule-based heuristics to rewrite a query into an equivalent execution plan with lower computation costs. For relational joins, MatRel can leverage the sparsity-preserving property of the merge function and Bloom-join strategies for efficient executions. Furthermore, MatRel adopts a cost model to generate communication-efficient matrix data partitioning schemes for input matrices on various join predicates. The experimental study on various applications demonstrates that MatRel achieves up to two orders of magnitude performance gain compared to state-of-the-art systems.

5 OPTIMIZING COMPLEX MATRIX-AWARE RELATIONAL QUERY EVALUATION PIPELINES – DEEP-LEARNING AS A DRIVING APPLICATION

5.1 Introduction

Recently, big data analytics has become more and more popular among both academics and industry. In the domain of Internet of Things (IoT)/Cyber-Physical Systems (CPS) [86, 87], deep-learning has become a dominant tool to study the complex relationships hidden in the large volumes of the observed data in both and academic research industry applications. It has been demonstrated that deep-learning exhibits potential to approximate complex datasets with high accuracy, significantly facilitating human-centered smart systems [88]. Compared to other ML models, deep-learning architectures can be adapted to various data types, e.g., video, audio, text, numerical, or combinations. A number of open-source platforms have been developed from major companies, Goolge's TensorFlow [80], Facebook's Pytorch [89], Amazon's MXNet [90], Microsoft's Cognitive Toolkit [91]. These carefully designed and well-tuned platforms further improve the performance of deep-learning models and make them easy to deploy for different use cases.

Deep-learning excels in supervised learning tasks for classification, achieving higher accuracy and better predictive performance than labor forces, such as handwriting and image recognition [92], speech recognition [93], and text understanding [94, 95]. For unsupervised learning tasks, deep-learning is leveraged for dimension reduction and density estimation. For instance, a high-resolution image contains tens of thousands pixels (features). Auto-encoders [96] are able to transform input data into an encoded output for compression. Furthermore, reinforcement learning exhibits potential for deep-learning without human supervision, through feedbacks from a connected environment. The ability of non-human interference has contributed significantly to

robotics and computer vision [97]. Google's AlphaGo [98] is the first deep-learning based intelligent agent to beat a professional human player without handicap[1]. Its successor, AlphaGo Zero [99] is able to learn without any human input, and significantly outperforms the prior implementations.

During the past few years, deep-learning has exhibited great development in feature learning of big data [100,101]. Compared to the conventional ML techniques, e.g., Support Vector Machine (SVM) and Naive Bayes [23], deep-learning models take advantage of numerous samples to extract the high-level features and learn the hierarchical representations by combining low-level inputs effectively. Therefore, deep-learning provides great potential to extract valuable knowledge from big data for prediction in industry and medical care [102]. In this chapter, we first give an overview of the deep-learning model. Next, we discuss the building blocks for composing computation pipelines of deep-learning. Finally, we illustrate our developed system, MatRel, is an easy-to-use and efficient platform for realizing deep-learning models, as a proof of concept.

5.2 Overview of Deep-learning Models

Many popular ML models can only accept inputs of certain data types, and have limited ability to learn complex data representations, e.g., Naive Bayes, linear regression, logistic regression, and decision trees. Deep-learning has originated from cognitive science, trying to imitate the learning process of human neurons and creating complex inter-connected neural structures. As pointed out by [103], the wide adoption of deep-neural networks is attributed to the fact that a generic neuron can be applied to any data type and learn a discriminative representation of the underlying data.

With recent advancement on large-scale distributed compute clusters, the implementation of large collections of neurons makes possible the neural networks. Though

[1] `https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol`

neural networks are ubiquitous today, they dated back to 1940s [103] and faded out of the focus of academics due to high complexity and compute deficiencies. However, the situation has clearly changed, thanks to the emerging applications where other ML models lag behind neural networks, such as image recognitions [92] and intelligent agents [99].

Essentially, deep-learning can be viewed as the application of multi-layer neural networks to conduct learning tasks. The basic computational neuron, the *sigmoid* neuron, is a single logistic activation function. Every neuron is linked to some input, and a loss function is leveraged to update the weights of the neuron and tune the logistic fit to the incoming data. When applying this procedure to multiple layers of many neurons, each neuron learns from all the outputs of the previous layers, reducing errors to produce an output. Therefore, the complexity is inevitably high for multiple inter-connected neurons.

Figure 5.1 illustrates the general structure for a deep neural network. The input layer ingests the raw data, and feeds them to multiple hidden layer neurons. Finally, the hidden layer neurons output the result, such as classification values or vectors that encode the raw input. The most basic layer is a fully-connected layer, where all neurons are linked to each other from the input layer. Alternatively, some links are randomly dropped in order to prevent over-fitting.

5.3 Required Operations for Deep-learning

Let us examine the required operations when evaluating the parameters of a deep neural network. Given a multi-layer neural network, the evaluation of an input is conducted by the *forward propagation* procedure. For an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$, the input to a hidden layer node can be computed as $z_i = \sum_k w_{i,k} * x_k + b_i$, where b_i is a bias term, or in terms of matrix algebra $\mathbf{z} = \mathbf{W} \times \mathbf{x} + \mathbf{b}$. Each hidden layer node has an activation function to transform the input z_i, e.g., a sigmoid function $\sigma(z_i) = \frac{1}{1+e^{-z_i}}$, or REctified Linear Unit (RELU) $f(z_i) = \max(0, z_i)$.

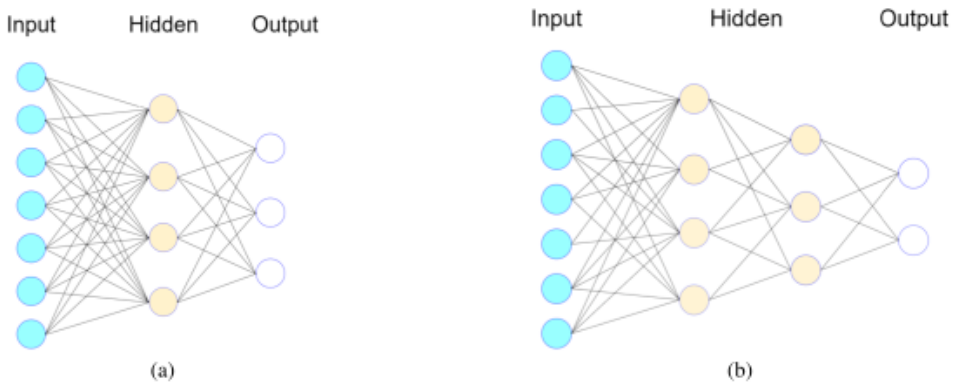


Figure 5.1.: Representative neural networks: (a) fully connected, and (b) including dropout [104].

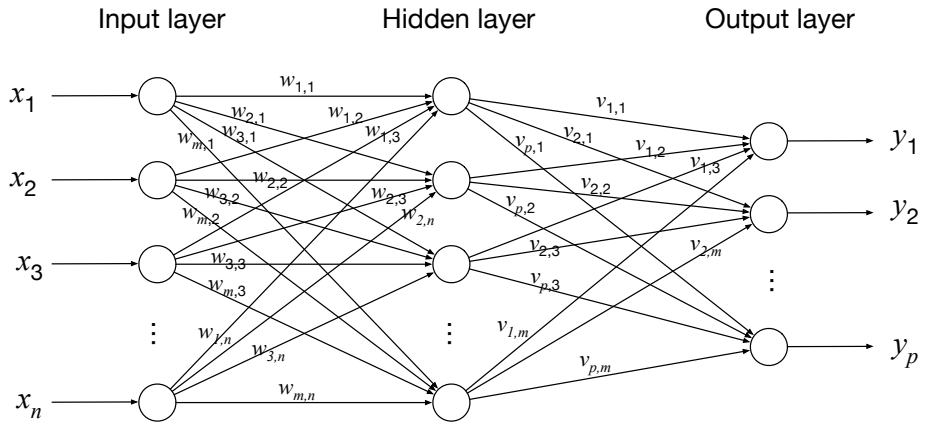


Figure 5.2.: Parameter learning of a general 3-layer neural network.

Thus, the output of the hidden layer can be evaluated as $\mathbf{h} = \sigma(\mathbf{W} \times \mathbf{x} + \mathbf{b})$, where the sigmoid function is chosen as the activation function. The output layer prediction can be computed in the similar manner as the hidden layer, $\mathbf{y} = \sigma(\mathbf{V} \times \mathbf{h} + \mathbf{c})$, where $\mathbf{c}$ is the bias vector of the output layer. Here, we assume the model parameters are already tuned, where matrix $\mathbf{W}$, $\mathbf{V}$ and vector $\mathbf{b}$, $\mathbf{c}$ are all known.

The training phase is more expensive, compared to the prediction phase. Given the training set $\{\mathbf{x}_i, y_i\}$, the objective is to minimize the loss function: $L = \frac{1}{2}\sum_i ||f(\mathbf{x}_i) - y_i||_2^2$, where $f(\mathbf{x}_i)$ is the prediction of $\mathbf{x}_i$ from the neural network, and y_i is the true

label of $\mathbf{x}_i$. Assume that there is only one node in the output layer, we organize all the unknown parameters in a single vector $\mathbf{u} = [w_{i,j}, v_i, b_i, c]^T$. The optimal solution of $\mathbf{u}$ is achieved when $\frac{\partial L}{\partial \mathbf{u}} = 0$. To obtain the partial derivative of L with respect to $\mathbf{u}$, we only need to find $\frac{\partial L}{\partial w_{i,j}}$, $\frac{\partial L}{\partial v_i}$, $\frac{\partial L}{\partial b_i}$, and $\frac{\partial L}{\partial c}$, and stack these partial derivative components together for $\frac{\partial L}{\partial \mathbf{u}}$. Leveraging the chain rule of partial derivatives, we can compute $\frac{\partial L}{\partial w_{i,j}} = \frac{\partial f}{\partial \mathbf{h}} \times \frac{\partial \mathbf{h}}{\partial w_{i,j}}$. The optimal parameters of $\mathbf{u}$ could be obtained by gradient-based numerical optimization methods, e.g., gradient descent method and stochastic gradient descent method.

As we have described so far, deep-learning models rely heavily on efficient matrix computations, which makes MatRel an ideal platform for training deep-learning models and making predictions. In addition, we show how to leverage relational data processing of MatRel to train deep-learning models on certain applications.

Let us look at the problem of learning word embedding (semantic representation) in natural language processing. There are two widely used Word2Vec models [105], Continuous Bag of Words (CBOW) and Skip-Gram. These two models essentially share the same model structure, and only differ in data dimensions of input and output layers. CBOW predicts the current word based on context, while Skip-Gram predicts surrounding words given the current word. To be specific, CBOW reads a word vector, and predicts a single word. On the other hand, Skip-Gram reads a single word, and predicts a word vector which most likely appear around the given input word. Both could be leveraged to learn the vector representation for each word in the corpus. For the purpose of demonstration, we choose Skip-Gram for discussion. Each input to Skip-Gram is a word in *one-hot* encoding, i.e., a vector $\mathbf{x}$ of length V (vocabulary size) and $x_i = 1$, $x_j = 0$ for $j \neq i$, which means *only* the i-th word appears in the input. The output is a set of surrounding words in one-hot encodings. The training samples are created by enumerating word pairs from a sliding window of a fixed size (say 10) on plain texts. In general, all the model parameters, e.g., each entry of matrix $\mathbf{W}$ and $\mathbf{V}$, need to be updated when a new training sample is encountered. This is an expensive procedure as there are millions of parameters to update for large datasets.

Negative sampling [106] is a technique that makes each training sample only update a small percentage of the weight matrix, rather than all of the entries. The relational selection is an ideal operator to choose which rows/columns of the weight matrix to update given the negative samples. Thus, MatRel is a perfect computation engine for Word2Vec models.

5.4 Query Interface

We extend MatRel's programming interface to allow composing deep-learning pipelines easily. Code 5.1 demonstrates the training procedure of the 3-layered neural network for Skip-Gram model in Scala. The corpus is loaded via the `loadWordPairs()` interface. The input word pairs are encoded with the one-hot encoder. The parameters are initialized with random floating point numbers. The neural work is configured with the `DNN` interface, where `sigmoid()` denotes the activation function. Gradient descent method is used for tuning the parameters, and the number of iterations is set to 100.

```
1  val data = loadWordPairs("in/pairs") // word pairs
2  val (X, ylabel) = oneHot(data) // one-hot encoding
3  val hSize = K // hidden layer size
4  var W = RandomMatrix(hSize, X.ncols) // weight
5  var V = RandomMatrix(X.ncols, hSize)
6  var b = RandomMatrix(hSize, 1L) // bias
7  var c = RandomMatrix(X.ncols, 1L)
8  val H = W %*% X.t + b
9  predict = DNN.sigmoid(V %*% H.t + c)
10 predict.run(100) // run 100 times
```

Code 5.1: Skip-Gram in Scala

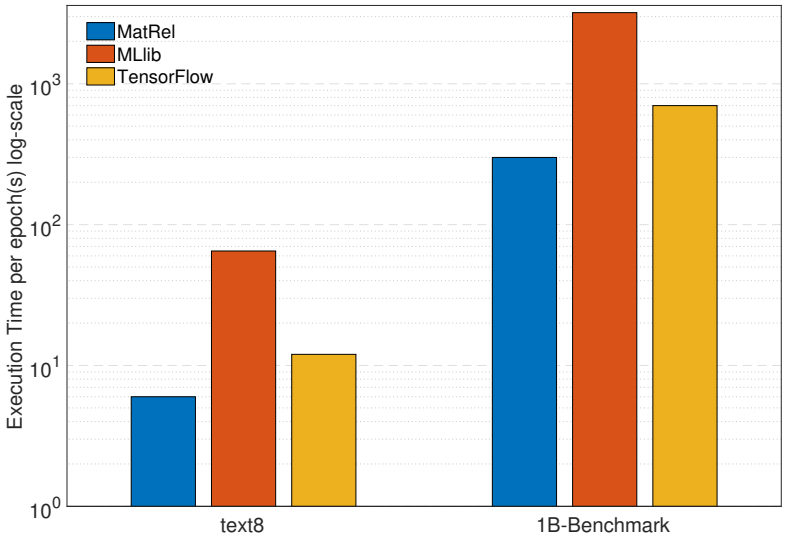


Figure 5.3.: Neural network training for Skip-Gram.

5.5 Performance Evaluation – Skip-Gram

We study the performance of MatRel for Skip-Gram on different datasets. The experiment setup is the same as the last chapter: an HP DL360G9 cluster with Intel Xeon E5-2660 realized over 6 nodes. The cluster uses Cloudera 5.9 consisting of Spark 2.1 as a computational framework and Hadoop HDFS as a distributed file system. Each node has 16 cores, 32 GB of RAM, and 400 GB of local storage. The total HDFS size is 1 Terabyte. Spark was configured with 5 executors, 16 cores/executor, 16 GB driver memory, and 24 GB executor memory.

The experiments are conducted on two real-world datasets: `text8`[2] and 1 billion word benchmark[3]. The `text8` dataset consists of 17 million words and 71K distinct words. The 1 billion word benchmark has 805 million words and 1.1 million unique words. We compare the runtime performance among MatRel, Spark MLlib, and TensorFlow on the two datasets.

[2] `http://mattmahoney.net/dc/textdata.html`
[3] `http://www.statmt.org/lm-benchmark/`

Figure 5.3 illustrates the execution time per epoch for three different systems. The Spark MLlib performs the worst, since it does not have a well-tuned library for sparse matrix computation. After one-hot encoding, the input matrix $\mathbf{X}$ is a sparse matrix. The labels also form a sparse matrix as each word is usually adjacent to a few words in the corpus. Without a delicate sparse matrix computation library, MLlib falls behind the other 2 systems. MatRel spends 282s on execution for each epoch on the 1 billion word benchmark, while TensorFlow takes about 700s. The performance gain of MatRel comes from the efficient query optimizer, which chooses the execution plan with lowest computation and communication costs.

5.6 Concluding Remarks

In this chapter, we discuss how MatRel could be leveraged to optimize complex matrix-aware relational query evaluation pipelines. Especially, we examine a popular family of ML models – deep neural networks. The core components required by deep neural networks are efficient matrix computations and relational operators. We demonstrate how to leverage MatRel for Word2Vec and show it achieves up to 2.5X speedups on real-world datasets, compared with the popular distributed deep-learning platforms.

6 CONCLUSION

In this dissertation, we study matrix-aware relational query processing in big data systems. With the emerging trend of in-database analytics, modern relational data systems are required to support complex statistical inference and machine learning models, where a number of these advanced tools rely heavily on linear algebra. Therefore, one question often arises, shall we develop systems specially for linear algebra support from scratch? This dissertation demonstrates that, we should leverage the well-tuned database query optimizer, and customize it to offer new opportunities for emerging application demands.

With application requirements rapidly evolving, our experiments demonstrates that the general database query optimizer needs to add new ingredients. For example, system without optimization, e.g., Spark MLlib, show worse performance than optimized ones. Thus, we believe that the developed techniques in this dissertation just open a tip of the iceberg for efficiently support linear algebra in a relational data system. For example, we propose and implement heuristic rule-based techniques for rewriting a matrix query when it possesses certain features. We develop cost models for optimizing sparse matrix multiplication chains and distributing matrix data partitions for reducing communication overhead. We propose a collection of transformation rules for reordering operators in a logical plan when both matrix and relational operators are present in the execution pipeline. We formalize the join operator on matrix data and develop a cost model for join execution on distributed matrix data over a cluster of machines. Overall, we believe these specific optimization strategies based on various characteristics of the query patterns and structures would enable fast innovation. Below, I present some open questions for future research:

- Better Support for Large Sparse Matrix Data: The combination of strong physical data independence and high-performance matrix computation for large sparse matrix data is ubiquitous in ML applications. Unfortunately, none of the existing big data systems support this requirement [107]. MatFast and its successor MatRel are one of the few systems provide some preliminary support. Spark MLlib has poor physical data independence and ScaLAPACK lacks support for sparse data. SystemML provides certain support for this combination, but it may be costly to maintain the metadata for input matrices in some scenarios. MatFast/MatRel maintains the sparsity for a matrix during the computation, but it may not be 100 percent accurate in certain cases. Therefore, more attention is needed for thoroughly supporting sparse matrix data for efficient and scalable matrix computation, including optimizing data layout, staging computations and communications etc.

- Memory Management for Distributed in-memory Matrix Data: Memory is scarce resource in a distributed computing system. A system needs to allocate the memory resource to different jobs based on their priorities and data access patterns. As a pioneering step, we have studied a special structure of matrix chain multiplications to reduce the sizes of intermediate results. However, more research work should be focusing on automatically determining the storage format of matrices for general computational patterns, especially when sparse matrices are present.

- Auto-tuning Matrix Data Layout and System Parameters: Most of existing big data systems require developers to decide the matrix data layout and system parameter during execution. For example, MLlib requires manually tunning partitioning schemes and caching mechanism. Such low-level decisions are cumbersome and unintuitive for ML-oriented end users. MatFast/MatRel 's optimizer offers automatically matrix data partitioning based on the cost model, but users still need to tweak the system parameters for Spark, e.g., caching

mechanism. Overall, more work is still needed to achieve full automation of determining matrix data layout and system parameters.

- Approximate Matrix Computations: Matrix computations are very expensive in terms of floating point number operations. By default, all the computations are conducted in double-precision to avoid overflow and losing significant digits in the result. However, many ML models do not require precise computation in each step. For example, an approximate solution during iteration is good enough to converge to the optimal solution in gradient descent. In certain cases, computations in single-precision lead to the similar result as those conducted in double-precision. The former has better compute efficiency and consumes less memory. However, this requires a deep understanding of the ML models and their computational paradigm.

REFERENCES

REFERENCES

[1] Y. Yu, M. Tang, W. Aref, Q. Malluhi, M. Abbas, and M. Ouzzani, "In-memory distributed matrix computation processing and optimization," in *ICDE'17*. Washington, DC, USA: IEEE Computer Society, 2017, pp. 1047–1058.

[2] Y. Yu, M. Tang, and W. G. Aref, "Scalable relational query processing on big matrix data," Purdue University, Technical Report, April 2018.

[3] Y. Yu, W. G. Aref, and M. Tang, "Matrel: A matrix-enabled relational data system," Purdue University, Technical Report, June 2018.

[4] M. Hilbert, "Big data for development: A review of promises and challenges," *Development Policy Review*, vol. 34, pp. 135–174, 2016.

[5] R. Han, X. Lu, and J. Xu, "On big data benchmarking," in *Big Data Benchmarks, Performance Optimization, and Emerging Hardware - 4th and 5th Workshops, BPOE 2014, Salt Lake City, USA, March 1, 2014 and Hangzhou, China, September 5, 2014, Revised Selected Papers*, 2014, pp. 3–18.

[6] S. Kuutti, S. Fallah, K. Katsaros, M. Dianati, F. Mccullough, and A. Mouzakitis, "A survey of the state-of-the-art localization techniques and their potentials for autonomous vehicle applications," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 829–846, 2018.

[7] Y. C. Tay, "Data generation for application-specific benchmarking," *PVLDB*, vol. 4, no. 12, pp. 1470–1473, 2011.

[8] J. Jiang, F. Fu, T. Yang, and B. Cui, "Sketchml: Accelerating distributed machine learning with data sketches," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15*, 2018, pp. 1269–1284.

[9] H. Itoh, A. Imiya, and T. Sakai, "Low-dimensional tensor principle component analysis," in *Computer Analysis of Images and Patterns - 16th International Conference, CAIP 2015, Valletta, Malta, September 2-4, 2015 Proceedings, Part I*, pp. 715–726.

[10] M. Weimer, A. Karatzoglou, and A. J. Smola, "Adaptive collaborative filtering," in *Proceedings of the 2008 ACM Conference on Recommender Systems, RecSys 2008, Lausanne, Switzerland, October 23-25*, 2008, pp. 275–282.

[11] "Linear regression," in *Encyclopedia of Parallel Computing*, 2011, p. 1033.

[12] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2011.

[13] S. Zhe, K. Zhang, P. Wang, K.-c. Lee, Z. Xu, Y. Qi, and Z. Ghahramani, "Distributed flexible nonlinear tensor factorization," in *Advances in Neural Information Processing Systems (NIPS)*, 2016.

[14] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *NIPS'01*, T. Leen, T. Dietterich, and V. Tresp, Eds. MIT Press, 2001, pp. 556–562.

[15] J. Bennett and S. Lanning, "The netflix prize," *KDD Cup*, pp. 35–35, 2007.

[16] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999.

[17] A. N. Langville and C. D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton, NJ, USA: Princeton University Press, 2006.

[18] N. A. Holtzman, P. D. Murphy, M. S. Watson, and P. A. Barr, "Predictive genetic testing: From basic research to clinical practice," *Science*, vol. 278, no. 5338, pp. 602–605, 1997.

[19] M. V Rockman and L. Kruglyak, "Genetics of global gene expression," vol. 7, pp. 862–72, 12 2006.

[20] J. Qi, H. Asl, J. Björkegren, and T. Michoel, "krux: matrix-based non-parametric eqtl discovery." *BMC Bioinformatics*, vol. 15, p. 11, 2014.

[21] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, 2013, pp. 3111–3119.

[22] O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," in *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, 2014, pp. 2177–2185.

[23] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[24] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *CoRR*, vol. abs/1402.3722, 2014.

[25] M. F. Goodchild, "Twenty years of progress: Giscience in 2010," *J. Spatial Information Science*, vol. 1, no. 1, pp. 3–20, 2010.

[26] V. Maliene, V. Grigonis, V. Palevicius, and S. Griffiths, "Geographic information system: old principles with new capabilities," *Urban Design International*, vol. 16, no. 1, pp. 1–6, 2011.

[27] X. Wang, A. Chowdhery, and M. Chiang, "Networked drone cameras for sports streaming," in *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, 2017, pp. 308–318.

[28] D. Albani, J. IJsselmuiden, R. Haken, and V. Trianni, "Monitoring and mapping with robot swarms for agricultural applications," in *14th IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS 2017, Lecce, Italy, August 29 - September 1, 2017*, 2017, pp. 1–6.

[29] K. Clarke, *Getting Started with Geographic Information Systems*. Upper Saddle River, NJ: Prentice Hall, 1997.

[30] "Hadoop," http://hadoop.apache.org/.

[31] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, no. 3, pp. 455–500, Aug. 2009.

[32] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[33] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04*. USENIX Association, 2004.

[34] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *CLOUDCOM'10*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 721–726.

[35] "Mahout," http://mahout.apache.org/, 2017.

[36] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "Systemml: Declarative machine learning on mapreduce," in *ICDE'11*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 231–242.

[37] "R," https://www.r-project.org/.

[38] M. Zaharia, *An Architecture for Fast and General Data Processing on Large Clusters*. New York, NY, USA: Association for Computing Machinery and Morgan, 2016.

[39] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "Mli: An api for distributed machine learning," in *ICDM*, 2013, pp. 1187–1192.

[40] "Mllib," http://spark.apache.org/mllib/, 2018.

[41] L. Yu, Y. Shao, and B. Cui, "Exploiting matrix dependency for efficient distributed matrix computation," in *SIGMOD'15*. New York, NY, USA: ACM, 2015, pp. 93–105.

[42] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[43] D. Kernert, F. Köhler, and W. Lehner, "Spmacho - optimizing sparse linear algebra expressions with probabilistic density estimation," in *EDBT*, 2015, pp. 289–300.

[44] M. Bohm, D. R. Burdick, A. V. Evfimievski, B. Reinwald, F. R. Reiss, P. Sen, S. Tatikonda, and Y. Tian, "Systemml's optimizer: Plan generation for large-scale machine learning programs," *IEEE Data Eng. Bull.*, vol. 37, no. 3, pp. 52–62, 2014.

[45] L. Muchnik, S. Pei, L. C. Parra, S. D. S. Reis, J. S. Andrade Jr, S. Havlin, and H. A. Makse, "Origins of power-law degree distribution in the heterogeneity of human activity in social networks," *Sci. Rep.*, vol. 3, May 2013.

[46] S. K. Thompson, *Sampling*, ser. Wiley series in probability and statistics. New York: J. Wiley, 2002.

[47] M. I. Jordan, Ed., *Learning in Graphical Models*. Cambridge, MA, USA: MIT Press, 1999.

[48] J. Nocedal and S. J. Wright, *Numerical optimization*, ser. Springer Series in Operations Research and Financial Engineering. Berlin: Springer, 2006.

[49] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[50] T. A. Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006.

[51] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI'12*. San Jose, CA: USENIX, 2012, pp. 15–28.

[52] "Blas," http://www.netlib.org/blas/.

[53] "Lapack," http://www.netlib.org/lapack/.

[54] "Scalapack," http://www.netlib.org/scalapack/.

[55] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *ICDM'09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 229–238.

[56] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang, "Madlinq: Large-scale distributed matrix computation for the cloud," in *EuroSys'12*. New York, NY, USA: ACM, 2012, pp. 197–210.

[57] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan, "Hybrid parallelization strategies for large-scale machine learning in systemml," *Proc. VLDB Endow.*, vol. 7, no. 7, pp. 553–564, Mar. 2014.

[58] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for large-scale machine learning," *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 960–971, Aug. 2016.

[59] P. G. Brown, "Overview of scidb: Large scale array storage, processing and analysis," in *SIGMOD'10*. New York, NY, USA: ACM, 2010, pp. 963–968.

[60] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: a scalable linear algebra library for distributed memory concurrent computers in frontiers of massively parallel computation," in *Symposium on the Frontiers of Massively Parallel Computation*, 1992.

[61] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016.

[62] C. Manning and D. Klein, "Optimization, maxent models, and conditional estimation without magic," in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology: Tutorials - Volume 5*, ser. NAACL-Tutorials '03. Stroudsburg, PA, USA: Association for Computational Linguistics, 2003, pp. 8–8.

[63] J. Beel, B. Gipp, S. Langer, and C. Breitinger, "Research-paper recommender systems: A literature survey," *Int. J. Digit. Libr.*, vol. 17, no. 4, pp. 305–338, Nov. 2016.

[64] F. Zhao, M. Xiao, and Y. Guo, "Predictive collaborative filtering with side information," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, 2016, pp. 2385–2390.

[65] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *SIGMOD '15*, 2015, pp. 1383–1394.

[66] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970.

[67] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009.

[68] A. Eldawy, M. F. Mokbel, and C. Jonathan, "Hadoopviz: A mapreduce framework for extensible visualization of big spatial data," *ICDE'16*, pp. 601–612, 2016.

[69] B. Huang, S. Babu, and J. Yang, "Cumulon: Optimizing statistical data analysis in the cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1–12.

[70] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, and S. Tatikonda, "Systemml: Declarative machine learning on spark," *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1425–1436, Sep. 2016.

[71] T. Elgamal, S. Luo, M. Boehm, A. V. Evfimievski, S. Tatikonda, B. Reinwald, and P. Sen, "SPOOF: sum-product optimization and operator fusion for large-scale machine learning," in *CIDR'17, 8th Biennial Conference on Innovative Data Systems Research*, 2017.

[72] R. Brijder, F. Geerts, J. V. den Bussche, and T. Weerwag, "On the expressive power of query languages for matrices," in *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*, 2018, pp. 10:1–10:17.

[73] M. Schleich, D. Olteanu, and R. Ciucanu, "Learning linear regression models over factorized joins," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 3–18.

[74] A. Kumar, J. Naughton, and J. M. Patel, "Learning generalized linear models over normalized data," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1969–1984.

[75] A. Kumar, J. Naughton, J. M. Patel, and X. Zhu, "To join or not to join?: Thinking twice about joins before feature selection," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 19–34.

[76] L. Chen, A. Kumar, J. Naughton, and J. M. Patel, "Towards linear algebra over normalized data," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1214–1225, Aug. 2017.

[77] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, "The multidimensional database system rasdaman," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '98. New York, NY, USA: ACM, 1998, pp. 575–577.

[78] E. Soroush, M. Balazinska, and D. Wang, "Arraystore: A storage manager for complex parallel array processing," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 253–264.

[79] J. Duggan, O. Papaemmanouil, L. Battle, and M. Stonebraker, "Skew-aware join optimization for array databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. ACM, 2015, pp. 123–135.

[80] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 265–283.

[81] P. Mehta, S. Dorkenwald, D. Zhao, T. Kaftan, A. Cheung, M. Balazinska, A. Rokem, A. Connolly, J. Vanderplas, and Y. AlSayyad, "Comparative evaluation of big-data systems on scientific image analytics workloads," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1226–1237, Aug. 2017.

[82] R. Zalipynis and R. Antonio, "ChronosDB: Distributed, file based, geospatial array DBMS," *PVLDB*, vol. 11, no. 10, pp. 1247–1261, 2018.

[83] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The madlib analytics library: Or mad skills, the sql," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1700–1711, 2012.

[84] S. Luo, Z. Gao, M. Gubanov, L. Perez, and C. Jermaine, "Scalable linear algebra on a relational database system," in *ICDE'17*. Washington, DC, USA: IEEE Computer Society, 2017, pp. 523–534.

[85] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang, "Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 681–690.

[86] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao, "A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1125–1142, 2017.

[87] A. Zanella, N. Bui, A. P. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22–32, 2014.

[88] N. D. Nguyen, T. Nguyen, and S. Nahavandi, "System design perspective for human-level agents using deep reinforcement learning: A survey," *IEEE Access*, vol. 5, pp. 27 091–27 102, 2017.

[89] "Pytorch," https://pytorch.org/.

[90] "Mxnet," https://mxnet.apache.org/.

[91] "The microsoft cognitive toolkit," https://www.microsoft.com/en-us/cognitive-toolkit/.

[92] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[93] D. Chen and B. K. Mak, "Multitask learning of deep neural networks for low-resource speech recognition," *IEEE/ACM Trans. Audio, Speech & Language Processing*, vol. 23, no. 7, pp. 1172–1183, 2015.

[94] N. Majumder, S. Poria, A. F. Gelbukh, and E. Cambria, "Deep learning-based document modeling for personality detection from text," *IEEE Intelligent Systems*, vol. 32, no. 2, pp. 74–79, 2017.

[95] Z. Jiang, L. Li, D. Huang, and L. Jin, "Training word embeddings for deep learning in biomedical text mining tasks," in *2015 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2015, Washington, DC, USA, November 9-12, 2015*, 2015, pp. 625–628.

[96] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[97] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 26–38, 2017.

[98] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[99] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, Oct. 2017.

[100] M. A. Alsheikh, D. Niyato, S. Lin, H. Tan, and Z. Han, "Mobile big data analytics using deep learning and apache spark," *IEEE Network*, vol. 30, no. 3, pp. 22–29, 2016.

[101] Y. Zhou, S. Zhao, X. Wang, and W. Liu, "Deep learning model and its application in big data," in *Design, User Experience, and Usability: Theory and Practice - 7th International Conference, DUXU 2018, Held as Part of HCI International 2018, Las Vegas, NV, USA, July 15-20, 2018, Proceedings, Part I*, 2018, pp. 795–806.

[102] X. Chen and X. Lin, "Big data deep learning: Challenges and perspectives," *IEEE Access*, vol. 2, pp. 514–525, 2014.

[103] I. J. Goodfellow, Y. Bengio, and A. C. Courville, *Deep Learning*, ser. Adaptive computation and machine learning. MIT Press, 2016.

[104] W. G. Hatcher and W. Yu, "A survey of deep learning: Platforms, applications and emerging research trends," *IEEE Access*, vol. 6, pp. 24 411–24 432, 2018.

[105] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[106] S. Stergiou, Z. Straznickas, R. Wu, and K. Tsioutsiouliklis, "Distributed negative sampling for word embeddings," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 2017, pp. 2569–2575.

[107] A. Thomas and A. Kumar, "A comparative evaluation of systems for scalable linear algebra-based analytics," *PVLDB*, vol. 11, no. 13, pp. 2168–2182, 2018.

VITA

VITA

Yongyang Yu has obtained an M.S. degree (2010), and a B.S. degree (2008) in the department of Computer Science and Technology from Harbin Institute of Technology (HIT), China. His research interests include database systems, machine learning, and matrix computations.